



NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

1989

CONTROL OF AN EXPERIMENT
TO MEASURE ACOUSTIC NOISE IN THE
SPACE SHUTTLE

by

Charles B. Cameron

June 1989

Thesis Advisor

Rudolf Panholzer

Approved for public release; distribution is unlimited.

Prepared for:
Naval Postgraduate School
Monterey, CA 93943-5000

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Dr. Harrison Shull
Provost

Reproduction of all or part of this report is authorized.

Code 39
Naval Postgraduate School
Monterey, CA 93943-5000

Reviewed by:

classified

Security classification of this page

REPORT DOCUMENTATION PAGE

Report Security Classification Unclassified		1b Restrictive Markings	
Security Classification Authority		3 Distribution Availability of Report Approved for public release; distribution is unlimited.	
Declassification/Downgrading Schedule		5 Monitoring Organization Report Number(s)	
Performing Organization Report Number(s) S62-89-022		7a Name of Monitoring Organization Naval Postgraduate School	
Name of Performing Organization Naval Postgraduate School	6b Office Symbol (if applicable) 39	7b Address (city, state, and ZIP code) Monterey, CA 93943-5000	
Address (city, state, and ZIP code) Monterey, CA 93943-5000		9 Procurement Instrument Identification Number Unfunded	
Name of Funding Sponsoring Organization Naval Postgraduate School	8b Office Symbol (if applicable)	10 Source of Funding Numbers	
Address (city, state, and ZIP code) Monterey, CA 93943-5000		Program Element No Project No Task No Work Unit Accession No	
Title (include security classification) CONTROL OF AN EXPERIMENT TO MEASURE ACOUSTIC NOISE IN THE SPACE SHUTTLE (Unclassified)			
Personal Author(s) Charles B. Cameron			
Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report (year, month, day) June 1989	15 Page Count 255
Supplementary Notes The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
Contract Codes		18 Subject Terms (include on reverse if necessary, and identify by block number)	
Id	Group	control. Space Shuttle, bubble memory, microprocessor, Get Away Special, autonomous, C, Z-80, acoustic, matched filter, Auxiliary Power Unit	
Abstract (include on reverse if necessary, and identify by block number) This thesis describes the potential use of a general-purpose controller autonomously to measure acoustic vibration in the Space Shuttle Cargo Bay during launch. The experimental package will be housed in a Shuttle Get Away Special (GAS) container. We have implemented the control functions with software written largely in the C programming language. We use an IBM PC DOS computer and C cross-compiler to generate Z-80 assembly language code, assemble and link this code, and then transfer it to EPROM for use in the experiment's controller. The software is written in a modular fashion to permit adapting easily to other applications. The software combines the experimental control functions with a menu-driven, diagnostic system to ensure that the software will operate in practice as it does in theory and under test. The experiment uses many peripheral devices controlled by the software described in this thesis. These devices include: a real-time data recorder, a bubble memory storage module, a real-time clock, an RS-232C serial interface, a power control system, a matched filter subsystem to detect activation of the Space Shuttle's auxiliary power units five minutes prior to launch, a launch detection subsystem based on vibrational and barometric sensors, analog-to-digital converters, and a heater system. The matched filter design is discussed in detail in this thesis, and the results of a computer simulation of the performance of its most critical sub-circuit are presented.			
Distribution Availability of Abstract Unclassified unlimited ☐ same as report ☐ DTIC users		21 Abstract Security Classification Unclassified	
Name of Responsible Individual Adolf Panholzer		22b Telephone (include Area code) (408) 646-2154	22c Office Symbol 72Pz

FORM 1473,84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

Control of an Experiment
to Measure Acoustic Noise in the
Space Shuttle

by

Charles B. Cameron
Lieutenant, United States Navy
B. Sc., University of Toronto, 1977

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
and
ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

ABSTRACT

This thesis describes the potential use of a general-purpose controller autonomously to measure acoustic vibration in the Space Shuttle Cargo Bay during launch. The experimental package will be housed in a Shuttle Get Away Special (GAS) canister.

We have implemented the control functions with software written largely in the C programming language. We use an IBM MS DOS computer and C cross-compiler to generate Z-80 assembly language code, assemble and link this code, and then transfer it to EPROM for use in the experiment's controller. The software is written in a modular fashion to permit adapting it easily to other applications. The software combines the experimental control functions with a menu-driven, diagnostic subsystem to ensure that the software will operate in practice as it does in theory and under test.

The experiment uses many peripheral devices controlled by the software described in this thesis. These devices include: a solid-state data recorder, a bubble memory storage module, a real-time clock, an RS-232C serial interface, a power control subsystem, a matched filter subsystem to detect activation of the Space Shuttle's auxiliary power units five minutes prior to launch, a launch detection subsystem based on vibrational and barometric sensors, analog-to-digital converters, and a heater subsystem. The matched filter design is discussed in detail in this thesis, and the results of a computer simulation of the performance of its most critical sub-circuit are presented.

REVISED
C.1925
C.1

TABLE OF CONTENTS

I. INTRODUCTION	1
A. GET AWAY SPECIAL (GAS)	1
B. THE VIBRO-ACOUSTIC EXPERIMENT	2
C. DIFFERENCES FROM EARLIER EFFORTS	2
1. Isolation of Microphones	2
2. Solid State Data Recorder (SSDR) Using Bubble Memory	3
3. Microprocessor Control of the Experiment	3
D. PROCEDURAL OUTLINE OF THE VIBRO-ACOUSTIC EXPERIMENT	4
1. Sweep Phase	4
2. Detection of the Auxiliary Power Units (APUs)	4
3. Scroll Phase	5
4. Launch Phase	5
5. Post-launch Operations	6
6. Abridged Experiment	6
E. IRREGULARITIES	7
F. OTHER APPLICATIONS	8
II. CONTROL HARDWARE	10
A. STANDARD CONTROLLER	10
1. NSC810A RAM-I O-Timers	10
2. On-board Analog-to-digital Converter	12
3. Bubble Memory Module for the Controller	12
4. Real Time Clock	13
5. RS-232C Serial Input Output Port	13
B. ADDITIONAL CONTROLLER HARDWARE	14
1. Analog-to-digital Converter Subsystems	14
2. Solid State Data Recorder (SSDR)	14
3. Matched Filter	16
4. Voltage Controlled Oscillator (VCO)	17
5. Vibration-activated Launch Detector	17
6. Barometric pressure switches	18

7. Heater Circuit	19
8. Power Control Subsystem	19
III. THE MATCHED FILTER	21
A. MICROPHONE INPUT STAGE	21
B. HIGH-PASS FILTER	22
C. PRE-AMPLIFIER	23
D. FOURTH-ORDER, ELLIPTICAL (CAUER), BANDPASS FILTER	23
E. ADJUSTABLE GAIN	32
F. FULL-WAVE RECTIFIER	32
G. LOW-PASS FILTER	34
H. THRESHOLD DETECTOR	38
I. RESETTABLE PULSE COUNTER	38
J. PULSE GENERATOR	40
K. SUMMARY	42
IV. DESIGN OF THE CONTROL SOFTWARE	43
A. MEMORY MAP	43
B. OPERATION OF THE VIBRO-ACOUSTIC EXPERIMENT	45
1. Menu-driven Diagnostic Program	45
2. Performing the Experiment	46
a. Microprocessor Control Program	46
b. Initialize Hardware	47
c. Run the Vibro-acoustic Experiment	47
d. Initialize Software	48
e. Do Sweep	49
f. Start Recording Response at Known Frequencies	51
g. Stop Recording Response at Known Frequencies	51
h. Wait for APU's to Start or for Launch Indications	53
i. Do Scroll	53
j. Abort	54
k. Do Launch	54
l. Check for a Completed Launch	55
m. Do Post-launch	57
n. Monitor Heater Subsystem Operation	57

o. Do Record	57
V. HOW TO GET THE EXPERIMENT READY FOR A LAUNCH	63
A. UNABRIDGED EXPERIMENT	63
B. ABRIDGED EXPERIMENT	64
C. BOTH VERSIONS OF THE EXPERIMENT	64
VI. TESTING OF THE SOFTWARE	65
VII. CONCLUSIONS	68
APPENDIX A. DERIVATION OF DESIGN EQUATIONS FOR THE MATCHED FILTER	72
A. BIQUADRATIC FILTERS USING TWO OPERATIONAL AMPLIFIERS	72
B. HIGH-PASS NOTCH FILTER	76
C. LOW-PASS NOTCH FILTER	78
D. A SECOND-ORDER, LOW-PASS FILTER USING ONLY ONE OPER- ATIONAL AMPLIFIER	80
APPENDIX B. CHOICE OF A SOFTWARE DEVELOPMENT SYSTEM	83
A. Z-80 ASSEMBLY LANGUAGE	83
B. CP/M AND TOOLWORKS C	83
C. MS DOS AND UNIWARE C	84
D. GENERATION OF FIRMWARE IN EPROM	85
APPENDIX C. HOW THE UNIWARE SOFTWARE USES THE COMPUTER MEMORY	86
APPENDIX D. HIERARCHICAL ORGANIZATION OF SOFTWARE FILES	88
A. SUBDIRECTORY \VIBRO\CONTRLR\HEADERS	88
B. SUBDIRECTORY \VIBRO\CONTRLR\CSOURCE	88
C. SUBDIRECTORY \VIBRO\CONTRLR\ASMSOURC	88
D. SUBDIRECTORY \VIBRO\CONTRLR\BATCH	88
E. SUBDIRECTORY \VIBRO\CONTRLR\LIST	89
F. SUBDIRECTORY \VIBRO\CONTRLR\OBJECT	89

APPENDIX E. SUBROUTINES, IN ALPHABETICAL ORDER BY NAME . .	91
--	----

APPENDIX F. SUBROUTINES, IN ALPHABETICAL ORDER WITHIN EACH MODULE	99
--	----

APPENDIX G. CONTROL PROGRAM DOCUMENTATION	107
---	-----

A. MAJOR SUBROUTINES AND FUNCTIONS	108
1. main()	108
2. void inithardware(void)	109
3. char checkprt(void)	111
4. void shut_down_no_log(void)	111
5. char menu(char experiment_flag)	111
6. void version(void)	113
7. void rtc(void)	113
8. void clockread(struct datetime *your_clock)	114
9. void dump_clock(struct datetime *clock)	114
10. void clockset(struct datetime *clock)	114
11. void testtimeout(void)	114
12. void pwrcnt(void)	115
13. void bubmenu(void)	115
14. char bub_on(void)	116
15. void bub_off(void)	116
16. char bubinit(void)	116
17. void bubcmdmenu(void)	117
18. void testpattern(char buffer[])	117
19. void showbubbuff(char buffer[], char mode)	117
20. char bubio(char command, int page, char *buffer)	118
21. void rdstatreg(void)	118
22. void expmnt(void)	118
B. SUPPORTING SUBROUTINES AND FUNCTIONS	121
1. File bubble.c	121
a. void bpageset(int page)	121
b. char issububcmd(char command)	124
2. File bubrw.s	125
a. char bubxfer(void)	125

b. char bubread(char *buffer)	125
c. char bubwrite(char *buffer)	125
3. File clock.c	126
a. void clockint(struct datetime *clock, struct idatetime *iclock) ...	126
b. char clockcompare(struct idatetime *clock1, struct idatetime *clock2)	126
c. void clocksum(struct idatetime *result, struct idatetime *clock1, struct idatetime *clock2)	126
d. void show_waketime(struct idatetime *waketime)	127
e. void dump_iclock(struct idatetime *clock)	127
f. void get_time(struct idatetime *clock)	127
g. void show_waketime(struct idatetime *waketime)	127
h. char timeout(int delaytime, int measure)	127
4. File convert.c	128
a. char atoh(char *ascii)	128
b. unsigned int atohexint(char ascii[])	128
c. int atoi(char *s)	128
d. char *bcd_asc(char bcd)	129
e. int bcd_int(char bcd)	129
f. char *ctoh(char byte)	129
g. char int_bcd(int decimal)	129
h. char *itoa(int n, char[])	129
i. char tolower(int c)	130
j. char *uitoh(unsigned int word)	130
5. File delay.s	130
a. void delay(int n)	130
6. File expmnt.c	130
a. char ad_read(char port)	130
b. int adtoint(char addata, unsigned long multiplier)	130
c. void alter_page0(struct page0data * pagezero)	131
d. char bad_idea_to_record(char show)	132
e. void display_page0(struct page0data * pagezero)	132
f. void do_sweep(void)	132
g. char initialize(void)	133
h. char listen(void)	133

i.	char logevent(char event)	133
j.	void log_menu(void)	134
k.	void monitor_heaters(void)	134
l.	void post_launch(void)	135
m.	void record(void)	135
n.	void short_experiment(void)	135
o.	void show_event(char event)	136
p.	void shut_down(void)	136
q.	char ssdrmode(char mode)	136
r.	char ssdr_status(void)	137
s.	char voltages_low(void)	137
t.	char we_launched(void)	137
7.	File fputc.c	137
a.	int fputc(int chr, void *device)	137
8.	File global.c	138
9.	File inout.c	138
a.	void allow_ctrl_interrupts(void)	138
b.	void dump(unsigned int address, unsigned int length)	138
c.	char gethex(void)	138
d.	unsigned int gethexint(void)	138
e.	int getint(void)	139
f.	int getpageno(void)	139
g.	char look_ahead(char *character)	139
h.	char termin(void)	139
i.	void testinput(void)	140
j.	void testoutput(void)	140
10.	File main.c	140
a.	void memory_dump(void)	140
b.	void testio(void)	140
11.	File mbrk.s	141
a.	char *mbrk(long size, long *realize)	141
12.	File newio.s	141
a.	char input(char port)	141
b.	void output(char port, char data)	141
13.	File power.c	141

a. void power_status(void)	141
b. char power_write(char command)	141
14. File start.s	142
C. PROGRAM MAINTENANCE	145
1. Procedures for Generating a New Executable Program	145
a. Compile the C source files	145
b. Assemble the Assembly Code Source Files	145
c. Link Modules Together	145
2. Getting the Executable Program into EPROM	146
a. Copy the Executable Program to a Diskette	146
b. Prepare to Write EPROMs	146
APPENDIX H. CONTROL PROGRAM SOURCE CODE	150
A. FILENAME SPEC	150
B. FILENAME VERSION.H	150
C. FILENAME VERSION.C	151
D. FILENAME VIBRO.H	151
E. FILENAME BUBBLE.H	158
F. FILENAME BUBBLE.C	158
G. FILENAME BUBRW.H	165
H. FILENAME BUBRW.S	165
I. FILENAME CLOCK.H	170
J. FILENAME CLOCK.C	170
K. FILENAME CONVERT.H	177
L. FILENAME CONVERT.C	177
M. FILENAME DELAY.H	181
N. FILENAME DELAY.S	181
O. FILENAME EXPMNT.H	182
P. FILENAME EXPMNT.C	182
Q. FILENAME FPUTC.C	199
R. FILENAME GLOBAL.H	200
S. FILENAME GLOBAL.C	200
T. FILENAME INITIAL.H	201
U. FILENAME INITIAL.C	201
V. FILENAME INOUT.H	203

W. FILENAME INOUT.C	203
X. FILENAME MAIN.H	210
Y. FILENAME MAIN.C	210
Z. FILENAME MBRK.S	213
AA. FILENAME NEWIO.H	214
AB. FILENAME NEWIO.S	214
AC. FILENAME POWER.H	215
AD. FILENAME POWER.C	215
AE. FILENAME START.S	217
AF. FILENAME ASM.BAT	220
AG. FILENAME ASMLIST.BAT	220
AH. FILENAME C.BAT	220
AI. FILENAME LINK.BAT	221
AJ. FILENAME LIST.BAT	221
AK. FILENAME LOADMAP.BAT	221
AL. FILENAME PRINTALL.BAT	221
AM. FILENAME PROMLINK.BAT	223
AN. FILENAME PROMOUT.BAT	223
AO. FILENAME PROMSYM.BAT	223
AP. FILENAME READYOUT.BAT	223
APPENDIX I. RS-232C INTERFACE PIN CONNECTIONS	224
LIST OF REFERENCES	229
INITIAL DISTRIBUTION LIST	231

LIST OF TABLES

Table 1.	ASSIGNMENT OF BITS IN THE RS-232C SERIAL INTERFACE PORT	14
Table 2.	BIT ASSIGNMENTS FOR READING POWER SUBSYSTEM RELAY SETTINGS	15
Table 3.	BIT ASSIGNMENTS FOR CONTROLLING POWER SUBSYSTEM RELAYS	16
Table 4.	SSDR COMMAND CODES	17
Table 5.	SSDR STATUS CODES	17
Table 6.	BIT ASSIGNMENTS IN PORT C ₁ OF NSC810A #1	18
Table 7.	BIT ASSIGNMENTS IN PORT C ₂ OF NSC810A #2	19
Table 8.	SUBROUTINE INDEX	91
Table 9.	MS DOS FILE INDEX	99
Table 10.	BIT ASSIGNMENTS FOR THE BUBBLE MEMORY CONTROLLER (BMC) STATUS BYTE	119
Table 11.	CONTENTS OF THE PARAMETRIC REGISTERS IN THE BUBBLE MEMORY CONTROLLER	122
Table 12.	CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\BATCH ..	143
Table 13.	CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\CSOURCE	146
Table 14.	CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\ASMSOURC	147
Table 15.	CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\HEADERS	148
Table 16.	RS-232C INTERFACE PIN CONNECTIONS	224
Table 17.	RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)	225
Table 18.	RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)	226
Table 19.	RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)	227
Table 20.	RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)	228

LIST OF FIGURES

Figure 1.	Block diagram of major components of the Vibro-acoustic Experiment.	11
Figure 2.	Block diagram of the Matched Filter.	22
Figure 3.	The microphone input stage.	23
Figure 4.	High-pass filter.	24
Figure 5.	Pre-amplifier.	25
Figure 6.	Magnitude of the transfer function of the elliptical bandpass filter.	26
Figure 7.	A generalized biquadratic filter using two operational amplifiers.	27
Figure 8.	A fourth-order, elliptic bandpass filter with $Q = 12$	28
Figure 9.	Notch filters	29
Figure 10.	Frequency response of the simulated bandpass filter	32
Figure 11.	Amplifier providing a variable voltage gain up to $28 = 28.9$ dB.	33
Figure 12.	Full-wave rectifier.	34
Figure 13.	A general second-order, single operational amplifier, low-pass filter.	35
Figure 14.	Second-order, low-pass filter.	36
Figure 15.	Threshold detector.	38
Figure 16.	Resettable Pulse Counter	39
Figure 17.	Astable operation of the LM555 Timer to generate a pulse train.	40
Figure 18.	Pulse Generator.	41
Figure 19.	Memory map of the computer	44
Figure 20.	Flowchart 0	48
Figure 21.	Flowchart 1	49
Figure 22.	Flowchart 2	50
Figure 23.	Flowchart 2.1	51
Figure 24.	Flowchart 2.2	52
Figure 25.	Flowchart 2.2.2	53
Figure 26.	Flowchart 2.2.4	54
Figure 27.	Flowchart 2.3	55
Figure 28.	Flowchart 2.4	56
Figure 29.	Flowchart 2.4.4	57
Figure 30.	Flowchart 2.5	58
Figure 31.	Flowchart 2.5.3	59

Figure 32. Flowchart 2.6	60
Figure 33. Flowchart 2.6.3	61
Figure 34. Flowchart 2.7	62
Figure 35. Hierarchical Organization of Software Files	89

GLOSSARY

Analog-to-digital (A/D) Converter: Analog signals are signals whose levels vary continuously as a function of time. Digital signals are signals which take on discrete (quantized) values, and these values remain constant for some given period of time, at which time the level is updated. An analog-to-digital converter samples a continuous input signal, decides which of a finite set of discrete values is the best one to describe the input signal, and outputs that discrete value. A regular clock is used to cause the input to be sampled again on a repetitive basis, and the output likewise is updated at the same rate. A digital computer cannot deal with continuous signal levels, so A/D converters are routinely used to let such computers read signal levels in the form they *can* handle, as digital values.

Auxiliary Power Unit (APU): The APUs are jet-turbine-powered engines used during both launch and recovery to operate the control surfaces of the space shuttle. Because they have a limited amount of fuel, the mission will be scrubbed if they operate for more than seven minutes before launch. The Vibro-acoustic Experiment attempts to detect them. If it is successful in doing so, it can anticipate launch.

ASCII: American Standard Code for Information Interchange. This is a seven-bit data code used in many digital systems to represent alphabetic and numeric characters, punctuation marks and a number of non-printing characters commonly used to pass information from one device to another. Since most digital systems are based on eight-bit bytes, one bit, the high-order one, is unused in the ASCII scheme. It is not uncommon for manufacturers to appropriate the extra bit for their own purposes.

BAUD: The baud rate is the number of symbols transmitted in one second. In many computer systems, one symbol can represent one bit (zero or one) and so the baud rate and the bit rate are equal.

BCD: Binary Coded Decimal. In this format, two four-bit codes are stored in a single eight-bit byte. Each of these four-bit codes can take on any of ten values from 0x0 through 0x9. Values from 0xa through 0xf are forbidden. The interpretation of these four-bit codes is that they represent the decimal digits from 0 through 9. Thus, a single eight-bit byte can represent decimal numbers from 0 through 99. This format is the only one used by the National Semiconductor MM58167A real time clock.

Bubble Memory: This is a form of integrated circuit memory which uses magnetic domains for storing information. These domains look like bubbles when viewed under a microscope, hence the name. Applying magnetic fields to the bubbles causes them to move about, permitting the information they represent to be stored and retrieved. From the standpoint of a user, they generally have two chief characteristics:

1. The data are stored in a combination of random and sequential methods. Thus groups of data can be accessed randomly, but the elements of the group must be accessed sequentially. This is analogous to the way a disk storage device operates. It accesses tracks directly, by moving its read-write head radially over the disk's surface to one of a set of concentric circles, called tracks. Once the head is positioned over the desired track, data is sequentially read from or written to it.

2. The data they contain are non-volatile. Removing power from them does not destroy their contents, provided this is done in a controlled manner. This is in contrast to the destruction of data in typical integrated circuit memories when power is removed from them. Those memories are non-volatile only if a battery backup is available. The Intel bubble memory we are using will lose data if the temperature wanders outside the range $[-20, +75]^{\circ}\text{C}$ [Ref. 1: Chapter 1, p.3].

Digital-to-analog (D/A) converters: See the earlier discussion of *analog-to-digital converters* for some background on the difference between analog and digital signals. The purpose of the digital-to-analog converter is to convert a digital signal to a smoothly varying continuous signal. Since the digital signal actually varies in jumps, it is not smooth to begin with. D/A converters use low-pass filters to eliminate the high-frequency components represented by the sudden jumps of a digital signal.

Dynamic: In the C programming language, most variables are *dynamic*. This means that they are created when a C function commences executing and are destroyed when that function completes executing. This is in contrast to the way *static* (*q.v.*) variables work.

EEPROM: Electrically erasable, programmable ROM (*q.v.*). The contents of EEPROMs are not as easily modified as are the contents of RAMs, but they are non-volatile (they don't lose their contents when power is removed.) The contents of these memories can be erased electrically, but generally at a much slower rate than that at which they can be read.

EPROM: Erasable, programmable ROM (*q.v.*). EPROMs can be erased for re-use if they are exposed to ultraviolet light for several minutes. It is usual to remove the integrated circuit from the circuit board to do this. EPROMs have a limited lifetime due to wear on the pins (unless zero-insertion-force sockets are used) and because their ability to be erased diminishes with age.

Executable Program Module: The output of the *link* process (*q.v.*) is a single file of machine code instructions. When placed in the computer's memory at the correct locations (specified in advance), these instructions permit the computer to execute a program.

FIFO: First-in, first-out. This term refers to a common data structure. One place this data structure is used is in the buffer on the bubble memory controller. That buffer serves as an intermediate storage area between the bubble memory and the user. For example, when data are being read from the bubble memory by the user, they are retrieved from the bubble memory by the bubble memory controller and placed in the FIFO buffer. Concurrently, data are being removed from the buffer and sent to the user. The first characters of information to arrive in the buffer are the first to leave, hence the first in are the first out.

Firmware: This term describes the computer programs which are stored in non-volatile memory, such as ROM (*q.v.*)

Handshaking: When two devices communicate, they employ a protocol which specifies which device does what, when. This protocol is referred to as "handshaking".

Hexadecimal: Numbers to the base 16. It is customary to use the usual digits (0-9) as well as the letters 'A' (or 'a') through 'F' (or 'f'), for the 16 distinct symbols required in this system. The C programming language by convention uses the prefix '0X' (or

'0x') to make it clear that the appended characters represent a hexadecimal quantity. For example,

$$2a_{16} \equiv 0x2a \equiv 2_{16} \times 16^1 + a_{16} \times 16^0 \equiv 2 \times 16^1 + 10 \times 16^0 \equiv 42.$$

Input/Output space: The Z-80 and the essentially similar NSC800 provide a separate set of addresses for input and output devices. Certain instructions are reserved for these addresses, which can run from 0x00 through 0xff. They do not interfere with the corresponding *memory address space* (q.v.)

I/O: Input or output.

I/O Space: See **Input/Output Space**.

Latch up: A comparator will ordinarily produce a high voltage when the non-inverting input receives a higher voltage than that present on the inverting input. Similarly, it will ordinarily produce a low voltage when the non-inverting input receives a lower voltage than that present on the inverting input. Some comparators are susceptible to the phenomenon called "latch up". This entails a failure of the comparator to change its output according to the usual rules. Instead, the output signal will remain stuck at one value without regard to changes at the input. This feature is highly undesirable, as it means that the comparator is no longer performing as it should.

Library: The output of the compilation or assembly steps is an object module. Several of these can be stored in a library for convenience. During the link process, the linker can look in the modules stored in the library for definitions of objects whose names it does not recognize. The alternative to putting modules in a library is to specify them individually to the linker, which is somewhat less convenient.

Linker: The linker is responsible for combining the object modules which comprise a complete program, and placing them in suitable memory locations. Object modules may include references to other modules or identifiers defined within other modules. These references must ultimately be resolved to memory addresses within the computer which will run the executable program. It is the job of the linker to perform this address resolution. To link a program is to request the linker to construct an executable program, and resolve all unknown addresses. The object modules may be obtained by the linker from either of two sources: from a library or from individual files containing only one module each. The output from the linker is a single file containing an executable program *module* (q.v.)

Memory address space: The Z-80 and the essentially similar NSC800 permit addressing memory with addresses in the range 0x0000 through 0xffff. Most instructions which use addresses, including stack instructions which do not explicitly address memory, use this space. There is another space of addresses called the *input output space* (q.v.)

Module: In the C programming language, many functions may be grouped together in a single file of source code. These are considered to comprise a single module, for they are compiled as a unit and the resultant object code is stored in a single file, an *object module* (q.v.). Similar remarks hold when the source code consists of assembly language instructions, rather than instructions in the C programming language. Indeed, this concept is applicable irrespective of the programming language used to create the executable program. There are several advantages to building modules in this fashion. Chief among them is the separation of sections of a program according to

their functional characteristics. This permits testing one module independent of testing any other module. It also facilitates the use of fully debugged programs for other applications at a later date.

Modulo: Consider a number x and another number m , called a modulus. The number x taken modulo m is written $x \bmod m$ and it is defined to be the least positive number n such that $x = k \times m + n$ for some integer k . As an example, $5 \bmod 6 = 5$ because $0 \times 6 + 5 = 5$. Similarly, $9 \bmod 6 = 3$ because $9 = 1 \times 6 + 3$, and $-2 \bmod 6 = 4$ since $-2 = -1 \times 6 + 4$. Although we can also write $-2 = -2 \times 6 + 10$, $-2 \bmod 6 \neq 10$ because 10 is not the least positive number which can be found to satisfy the equation.

Nibble: A nibble is a half byte. This is a typical example of humor in the computer business.

NSC810A: An integrated circuit from National Semiconductor which includes two eight-bit ports, one six-bit port, 128 eight-bit words of RAM (*q.v.*) and two 16-bit binary timers.

Object Module: An almost-executable computer program. The reason it is not fully executable is that not all addresses within it have been resolved yet, nor has the linker established what addresses should be assigned to relocatable programs. Assemblers and compilers produce object modules. Linkers convert them into executable form by resolving the unresolved addresses and assigning all relocatable code to its final location.

Parametric Registers: The Intel BPK 5V75A Four-Megabit Bubble Memory includes five parametric registers which must be loaded prior to attempting to perform input from or output to the bubble memory. Two of the five comprise the block length register, which defines both the number of bytes contained in a page of bubble memory (*e.g.*, 64), and the number of pages to be transferred from bubble memory to the bubble memory port or *vice versa* at a time. Two more specify at which of the 8,192 pages in the bubble memory to start the transfer of data. The last, the "enable" register, primarily defines whether operation is to be interrupt-driven or not.

Project G-313: This is the designation of the NASA project comprising the Vibro-acoustic Experiment.

PROM: Programmable ROM (*q.v.*) These ROMs can be written to once by the user, but once written, their contents can never be modified.

Quotation Marks: In C, double quotation marks (" ") are used to enclose character strings. Internally, the C compiler always places an ASCII NUL character (its hexadecimal representation is 0x00) at the end of a string. Single quotation marks (' ') are used to enclose a single character. Internally, the C compiler does *not* append an ASCII 0x00 to a single character.

RAM: Random access memory. This refers primarily to memory which can be written to and read from repeatedly. It commonly is volatile, *i.e.*, its contents are destroyed when power is removed.

ROM: Read-only memory. This term is a bit of a misnomer. Obviously a memory which can never be written to would be of little value. Generally, it is much more difficult to modify the contents of a ROM than it is to modify the contents of a RAM.

ROMs come in several varieties:

1. A mask-programmed ROM receives its data at the factory according to a customer's specification when it is manufactured.
2. A PROM (*q.v.*) is programmed once by the user.
3. An EPROM (*q.v.*) can be programmed repeatedly, but must be erased by ultra-violet light between uses.
4. An EEPROM (*q.v.*) also can be programmed repeatedly, but it can be erased electrically.

RS-232C Serial Interface: This interface is also known as the *EIA standard interface*. It was developed in 1969 by the Electronic Industries Association in conjunction with the Bell system, as well as independent manufacturers of computers and modems. Data are transmitted serially using two voltage levels. $+V_s$ represents a binary 0; $-V_s$ represents a binary 1.

The voltage V_s can lie within the range [3,25]V. While the RS-232C defines the electrical characteristics of the interface, the functional description of the interchange circuits, and lists standard applications, it is silent on the subject of physical connectors. Usually, however, DB-25 connectors having 25 pins are used. The tables in APPENDIX I. RS-232C INTERFACE PIN CONNECTIONS on page 224 show the pin connections for the RS-232C interface. [Ref. 2; p. 683]

SSDR: Solid State Data Recorder. This device stores audio data in magnetic bubble memories. It accepts commands analogous to those selected by pushing a button on a conventional, reel-to-reel tape recorder. For example, the commands **PLAY** and **RECORD** exist. However, access to the data can be random.

Static: In C, most variables are *dynamic* (*q.v.*). They can be made *static* by the inclusion of this keyword in their declarations. This causes them to become permanent. They are not then created when the function in which they are declared starts to execute. They are created at the time of compilation. They do not lose their contents when that function's operation ends. The contents of the storage locations assigned to them remain intact until the next time that function tries to access that variable.

UART (Universal Asynchronous Receiver-Transmitter): A common integrated circuit which provides asynchronous communications between two hardware devices. We use it to implement an RS-232C serial interface between the controller hardware and a terminal.

Volatile Memory Storage: Conventional RAM (*q.v.*) loses its contents when power is removed. This property is called volatility. By contrast, magnetic core and bubble memories are non-volatile. For that matter, printed pages are also non-volatile memories.

Voltage Controlled Oscillator (VCO): The VCO operates a loudspeaker in the experiment during the *sweep* phase. The frequencies are increased incrementally between 35 Hz and 785 Hz in 1 Hz increments.

ACKNOWLEDGEMENTS

This thesis would never have been written without the steady support and encouragement given me from the very beginning by my wife, Diane. For this I am forever indebted to her. I must also give special thanks to Professor Rudy Panholzer, who guided my efforts from the beginning; Professor Steve Garrett, for teaching me about analog electronics; Mr. David Rigmaiden, for tireless advice on technical matters; CPT Frank Mazur, USMC, for taking me under his wing at the start of my involvement in the project; CAPT Ron Byrnes, USA, without whose patient and calm help the bubble memory still would not be working properly; Mr. Larry Frazier, whose vast knowledge of Script, GML, and GThesis, and continual willingness to answer any and all questions about them, were of enormous help to me; Dr. Otto Heinz for his guidance throughout my course of study; CDR Steve Hannifin, USN, and CDR Skip Braden, USN, my former Commanding Officers, who stood by me when I needed them; and last, but far from least, to the United States Navy and the people of the United States of America for permitting me the great privilege of studying at the Naval Postgraduate School.

I. INTRODUCTION

Since it began flying into space in April, 1981 the Space Shuttle has made it much easier to get payloads into space. Although the shuttle was grounded because of the tragic explosion of the Challenger on January 28, 1986, flights resumed in October, 1988.

A. GET AWAY SPECIAL (GAS)

In 1976 the National Aeronautics and Space Administration (NASA) established the Get Away Special (GAS) program [Ref. 3: p.11]. The purpose of this program is to permit individual experimenters to have room on the Space Shuttle for their experiments, provided there is no undue interference with the rest of the mission as a result. To preclude such interference, NASA therefore imposes a number of constraints on these experiments. Among these are

1. They must contain their own power source, heating, data handling facilities, and so on [Ref. 4: p.8].
2. No more than three external switches may be provided for operation by crew members. Of these, one must be devoted to removing all power from the payload [Ref. 4: p. 28].

The Space Shuttle is subject to powerful acoustic vibrations during launch. In the past, minor breakage of crystals and circuit boards has resulted [Ref. 3: p.11]. It is thought that the vibrations are responsible for this damage, and that some regions of the cargo bay are more susceptible to damage than others. Acoustical analysis of the sound waves which cause these vibrations could reveal where the best and worst locations are.

Several early GAS experiments carried conventional reel-to-reel tape recorders and were intended to record the acoustic waves in the cargo bay for subsequent analysis. However, the analysis was flawed for several reasons [Ref. 5: p. 15]. Among these were:

1. The microphones were mounted close to the bulkhead of the cargo bay, within a partial enclosure. Thus the data might have been erroneous.
2. The data recorded by the microphone may have been contaminated by interaction between the microphone and its isolation system.
3. The acoustical waves in the forward third of the cargo bay were not recorded.

Furthermore, astronauts were too busy at launch time personally to initiate the experiments. Instead, the experiments included circuitry to detect the roar of the main engines and trigger the commencement of the experiment [Ref. 6: p. 11]. There is good reason

to doubt the validity of the analysis of acoustic waves whose collection was itself triggered by the occurrence of those same waves. As a result of all these factors, the analysis so far has been ambiguous.

B. THE VIBRO-ACOUSTIC EXPERIMENT

The Space Systems Academic Group at the Naval Postgraduate School plans to conduct an experiment as NASA project G-313 to obtain improved acoustical measurements in the cargo bay of the Space Shuttle during launch. In the remainder of this thesis, we shall refer to this project as the Vibro-acoustic Experiment. The reader is referred to [Ref. 7] for a general overview of the experiment.

The purpose of this thesis is to describe the software and some of the hardware which controls the Vibro-acoustic Experiment. At times this thesis will merely describe the work we have done. At other times, it will prescribe what to do to achieve various ends. Thus it will serve not merely as documentation of what has been done, but it will also serve as a manual for those who might wish to elaborate on this earlier work.

The majority of the work performed by the author had to do merely with the control of the experiment. However, the *matched filter* (described in Chapter III. THE MATCHED FILTER on page 21) was redesigned by the author and is completely described in this thesis.

A great deal of the hardware and software created to control the Vibro-acoustic experiment is very general in nature, and would apply without change to other experiments. We will attempt to indicate which components have general applicability. The hope is that future applications will benefit from this approach, and will be spared the need to build and program a controller from scratch. More information on the general-purpose controller hardware we use in the Vibro-acoustic Experiment can be found in Chapter II. CONTROL HARDWARE on page 10. The software is described in general terms in Chapter IV. DESIGN OF THE CONTROL SOFTWARE on page 43.

C. DIFFERENCES FROM EARLIER EFFORTS

Like earlier experiments, this one is housed in a GAS canister. It differs from them, however, in several key respects.

1. Isolation of Microphones

The experiment uses microphones housed in a mounting designed at the Naval Postgraduate School to isolate them from vibration. This is intended to reduce the

contamination of the recorded acoustical waves by structural vibrations. This microphone arrangement is described in Stehle [Ref. 5].

2. Solid State Data Recorder (SSDR) Using Bubble Memory

Conventional reel-to-reel tape recorders are supplanted by a recorder using magnetic bubble storage. This recorder was also designed at the Naval Postgraduate School and has the following advantages over conventional recorders.

1. It contains no moving parts, so is less prone to mechanical failure.
2. It permits random access to data, not possible with tape. While such a capability is commonplace with disk storage, disks do suffer from mechanical breakdown. Also, they are vulnerable to errors when external accelerations occur, as they do during a launch. This is less problematic in the case of bubble memories.
3. Bubble memory is non-volatile, that is, its contents are not destroyed when power is removed. Thus battery power is not required to keep the stored data available. Slightly offsetting this advantage is the fact that power must be removed in a controlled fashion, and specified temperature limits must be maintained.

The magnetic bubble recorder is described in Frey [Ref. 8].

3. Microprocessor Control of the Experiment

To operate the experiment, another group at the Naval Postgraduate School built a single-board, microprocessor-based controller. This general-purpose controller uses a National Semiconductor NSC800 microprocessor (roughly equivalent in function to a Zilog Z-80). This controller, as it was originally conceived, is described in Wallin [Ref. 3]. From a programmer's standpoint, the controller has the characteristics described in Chapter II, Section A. Standard Controller on page 10.

The controller will be responsible for:

1. activating all subsystems at the appropriate time;
2. monitoring execution of the experiment;
3. keeping a log of significant events and the dates and times at which they occurred. This log is stored in the controller's bubble memory module;
4. recording temperature and voltage readings while the shuttle is in space; and
5. ensuring that the bubble memories do not get too cold. This is done by intermittently operating the heater subsystem to maintain a temperature above 10°C. [Ref. 1: p. 3]

In addition to the obvious functions called for in controlling the experiment, the software also contains a menu-driven diagnostic subsystem to provide for testing on the ground. (See APPENDIX B. CHOICE OF A SOFTWARE DEVELOPMENT SYS-

TEM on page 83 for a general description of the several software development systems we have used.)

D. PROCEDURAL OUTLINE OF THE VIBRO-ACOUSTIC EXPERIMENT

In this section we sketch an outline of the operation of the Vibro-acoustic Experiment. The flowcharts in Chapter IV, Section 2. Performing the Experiment on page 46 show the procedure which the experiment follows. A synopsis of this procedure is provided here.¹ The experiment begins to operate when the ground crew or astronauts turn on a switch in the cabin, causing power to be applied to the GAS canister which houses the experiment. With power applied, the microprocessor comes to life. Its first task is to initialize the programmable hardware ports and timers. It then has to decide whether or not to perform the complete experiment or an abridged version of it. The need for such a decision will become apparent presently. For the moment we will confine our attention to the unabridged experiment.

1. Sweep Phase

Once the cargo bay has been loaded, the ground crew will activate the experiment for about an hour to let it perform the *sweep* phase. During this phase, the cargo bay is irradiated with a sequence of acoustic tones of known frequencies and the acoustic response of the enclosure is recorded by the Solid State Digital Recorder (SSDR). After the mission, analyzing this data [Ref. 9] and comparing it to the echoes recorded during launch will reveal the locations of the regions most and least prone to damage from vibration. This phase is the longest, and lasts 13 minutes.

2. Detection of the Auxiliary Power Units (APUs)

The Space Shuttle's Auxiliary Power Units (APUs) are jet turbines used to operate control surfaces during launch and recovery of the shuttle. The APUs start to operate around five minutes before launch. Because they emit a characteristic frequency at 600 Hz, we can use a matched filter to detect their acoustic signature [Ref. 6: pp. 15-18]. When the matched filter detects this signal, it knows that launch is imminent, and it is time to start recording the sounds which occur prior to launch. Thus we will record the sounds before, during, and after launch. By not waiting for the roar of the main rocket engines before starting to record the ambient noise, we will avoid the problem mentioned in Chapter I, Section A. Get Away Special (GAS) on page 1. The data collected by this means should be much more accurate.

¹ See Chapter IV, Section B. Operation of the Vibro-acoustic Experiment on page 45 for complete details.

Since there exists the possibility that for some reason the matched filter will fail to detect the APU's, the experiment includes two backup systems.

1. The Vibration-activated Launch Detector will detect the vibrations associated with launch.
2. A second backup system will use two barometric pressure switches to detect the drop in atmospheric pressure which occurs as the Space Shuttle rises. These switches will be placed in a redundant, parallel configuration. Thus, if either one or both of them work, the drop in barometric pressure will be detected.

Neither of these systems can detect operation of the APU's. However, if either one should detect a launch, the control program stops waiting for the APU's to come on and switches immediately to the *launch* phase.

If the matched filter successfully detects the APU's but the Vibration-activated Launch Detector fails to detect launch, the barometric sensor will cause the experiment to switch to *launch* phase, albeit a little late.

It would be unfortunate if the matched filter failed to detect the APU's, for then one of the primary advantages of the Vibro-acoustic Experiment over earlier efforts to record acoustical noise in the Space Shuttle would disappear. It would be doubly unfortunate if *neither* the matched filter *nor* the vibration detection subsystem worked, for then no data would be recorded until well after launch. If any one of the three systems works as designed, then the experiment will acquire at least *some* data.

3. Scroll Phase

If the matched filter detects the APU's, the control program will place the Solid State Data Recorder (SSDR) into *scroll* mode. In this mode, the SSDR uses a subset of its bubble memory for recording the ambient noise prior to launch. The fraction of memory dedicated to this purpose permits at most 110 seconds of recording time. Once this memory is used up, the SSDR will start re-using it from the beginning.² As a result of this mode of operation, roughly two minutes of pre-launch noise will be recorded, along with the noise of the ignition of the main engines.

4. Launch Phase

Either the vibration detection subsystem or the barometric sensors can trigger detection of a launch. When a launch is detected, the control program puts the SSDR into *launch* mode. The purpose of this mode is to record the noise after the launch begins. In *launch* mode, enough memory is dedicated to permit about two minutes of

² In effect, the SSDR is writing onto a continuous, looped scroll, hence the name of this mode of operation.

ambient noise to be recorded.³ Once this memory is exhausted, the SSDR will signal the control program that it has finished operations.

5. Post-launch Operations

After the SSDR has signalled completion, the Vibro-acoustic Experiment has finished gathering all the acoustical data it needs. The controller will continue, however, to monitor and record temperature in the GAS canister and record this information in its own bubble memory module. It will also monitor and record voltage levels of each of three power supply batteries. If these should fall below 8.5V, it will halt. This prevents loss of data in the bubble memory due to insufficient voltage and current levels.

The basis for choosing 8.5V follows. Each individual cell is rated for 2.0V. There are five of these cells in the 10 V stack which powers the bubble memory. If any cell drops to 1.81V or below, it is considered to be below the operating threshold. Five such cells generate 9.05V. The bubble memory itself can operate on as little as 5 V. We know that the batteries are losing power when the voltage falls below 8.5 V, but we still have a margin of 3.5 V above the voltage required to operate the bubble memory. (The margin is reduced slightly due to the presence of 5 V voltage regulators in the circuit, but it is still ample.) It is therefore reasonable to halt operations if the voltage falls below 8.5V.

Also, if the temperature of the bubble memory falls below 10°C, it is below the minimum operating temperature, and we will suspend operation of the bubble memory until the temperature returns to 10°C once more. Likewise, if the temperature should rise above 55°C, it is above the maximum operating temperature, and bubble memory operations will be suspended until the temperature drops within the operating range again.⁴ [Ref. 1: Chapter 1, p. 3]

6. Abridged Experiment

NASA has balked at the idea of our performing the *sweep* phase. They are concerned over the possibility that the loud sounds generated during the *sweep* might damage other payloads or frighten technicians. They also are reluctant to remove per-

³ Since the shuttle's cargo bay is not pressurized, the air will leak out as the outside pressure drops. After two minutes, there will be no appreciable atmosphere left inside the cargo bay, and all sound will have ceased.

⁴ These checks are only performed during the post-launch phase, which begins within two or three minutes from the time of launch. It is unlikely that NASA would launch the shuttle if the outside temperature were below 10°C, and we do not anticipate that the temperature will fall appreciably within the first three minutes of flight.

sonnel from the vicinity of the shuttle during that phase. They are equally reluctant to require those personnel to wear hearing protection during the *sweep*.

Those arguments seem specious to us. It strikes us as unlikely that damage to equipment might result from the *sweep* but not from the rocket motor noise during lift off. We cannot see the reason why personnel whose hearing might be damaged during the *sweep* cannot wear hearing protection. While we can still perform an analysis of the recorded data without first doing a *sweep*, it is likely to produce less useful results.

We have decided to proceed as we wanted to originally, that is, to design an experiment which would do everything we want. We have added an additional decision point to permit the abridged experiment to take place. This shortened experiment would simply turn on the recorder when the APU's were detected or when launch occurred, and we would hope for the best. There would be no *sweep*, no *scroll*, and no *launch* phases: there would only be a *record* phase, once the APU's or a launch were detected.

Once NASA sees that the abridged experiment works, that the analysis provides good results, and that permitting the unabridged experiment will yield even better data, we hope that they will relent and permit us to fly the experiment again in the unabridged mode.

E. IRREGULARITIES

What happens if the power fails temporarily and then is restored? This might happen if the power switch is inadvertently switched off by the ground or flight crew, or through some equipment malfunction. Upon the restoration of power, the micro-processor must decide where in its procedure to resume execution. There are several cases to consider.

1. The *sweep* phase has never been initiated, nor has a launch occurred. The correct action is to start at the beginning.
2. The *sweep* phase has been initiated. It is not known whether or not it ever was completed, but a launch has not occurred. The correct action is to skip the *sweep* phase and wait for some indication of a launch. The *sweep* creates a very loud noise which would be hazardous to ground personnel if it were permitted to occur at other than a scheduled time. Since this time is not known at present, and never is firmly enough known in advance to be programmed into the computer, we cannot risk running the *sweep* phase if it is interrupted by a power fault.
3. The *sweep* phase has been initiated (and presumably completed), the APU's are on, but no indications of a launch are present. The correct action is to enter *scroll* mode. If it was already in progress when the power fault occurred, it will be re-started. This is all right, since no vital information will be lost by this procedure.
4. The *sweep* phase has been initiated (and presumably completed) and conditions of a launch are present but the barometric switch has never been tripped. The correct

action is to assume we are just beginning a launch and to initiate *launch* mode. This creates a risk that recordings of the moment of launch would be lost if a power fault occurred between the moment of launch and the triggering of the barometric switch as the spacecraft ascends. There is no obvious way entirely to eliminate this risk.

5. The *sweep* phase was initiated (and presumably completed) and the barometric switches were activated at some earlier point. This implies that the power fault took place after the activation of the barometric switches, that is, after launch. The correct action is to assume that launch data was successfully recorded and to initiate the post-launch monitoring operations.

F. OTHER APPLICATIONS

The controller hardware is sufficiently powerful that it could easily provide control for other applications. In particular, many spaceborne applications could be operated by it.

In the course of developing the control software, we had to create support routines to take care of a great many mundane functions. In computers with operating systems, these functions are typically provided by the operating system. The user has merely to know about them and use them.

The controller we use *has* no operating system, so we had to create many low-level functions, *e.g.*, one which converts a hexadecimal number to a character string representing that number.

At a higher level, we wrote subroutines to display text on a terminal, operate a bubble memory, operate a real-time clock, and control various external devices through the 44 input and output lines provided with the controller.

By using the low-level subroutines, and by organizing an application's software in a similar manner, much of the most tedious and uninspiring work entailed in producing a control program for this controller hardware could be avoided. This would leave more time to devote to the real purpose of the application. In an environment with few people available to do the work, such economy is very attractive.

Another consideration is the lack of a need to use assembly language in programming the controller. In the very few places where it was required, we used it. Along with the large collection of C language subroutines of general applicability, the routines we have already provided in assembly language should suffice for almost all run-of-the-mill work.

In one area we were ourselves compelled to abandon the use of C language source code and switch to assembly language code. We initially hoped that only the start-up code, the input routine, the output routine and the software delay routine would require

the use of assembly code. We assumed we could input from and output to the bubble memory using compiled C language source code.

This assumption turned out to be incorrect. We needed a data transfer rate of 16,000 bytes per second [Ref. 1: Chapter 1, p. 3], but could only attain around 3,000 bytes per second. Consequently, we had to replace a small section of C code with assembly language source code. Even this was necessary only because we used a *prototype* bubble memory board with a buffer whose size was inadequate to handle data transfers. While this buffer provided only 40 characters of space, we needed 64.

When speed becomes paramount, assembly code may be necessary, since it can be tailored to the job at hand and so produce very efficient programs.⁵ For many applications, however, speed is not critical. It ordinarily is foolish to waste time achieving in assembly language what can be done much more quickly using a high-level language. Only if you *cannot* achieve the desired performance with a high-level language, must you use assembly language.

Irrespective of whether some portion of an application does or does not demand efficient code (written in assembly language), for most applications the majority of the code can be written in a high-level language such as C. Many applications, too, need no more facilities than those provided in the Vibro-acoustic Experiment. In such cases, building on the work presented in this thesis has very clear advantages.

⁵ Nonetheless, some optimizing compilers surpass quite competent assembly language programmers in efficiency.

II. CONTROL HARDWARE

The controller we use in the Vibro-acoustic Experiment is based on the NSC800 microprocessor. For all practical purposes, this is functionally equivalent to the Zilog Z-806 [Ref. 10]. Figure 1 on page 11 is a block diagram showing the major components of the system. To the left of the microprocessor appear those peripherals which ordinarily fall under the control of the standard controller. We discuss these peripherals and their capabilities in Section A. Standard Controller below. One can connect an assortment of devices to the 44 input and output lines available on it. Other applications than the Vibro-acoustic experiment could use this bare-bones controller for their own purposes.

To the right of the microprocessor appear those peripherals which are peculiar to the Vibro-acoustic Experiment. We discuss these peripherals and their capabilities in Section B. Additional Controller Hardware on page 14 below.

A. STANDARD CONTROLLER

1. NSC810A RAM-I/O-Timers

Two NSC810A RAM-I O-Timer units provide four eight-bit ports and two six-bit ports. These provide 44 bits of input and output capability. There also are two timers on each device. One of each pair is completely independent of the data ports; the other, if used, reduces the number of available pins in the six-bit port to three. We have configured the system such that one of the latter kind of timer is unavailable, since we have dedicated the data lines with which it interferes to other purposes. The two timers which do not conflict with any data lines at all are dedicated to providing:

1. A 153.6 kHz signal to the IM6402 Universal Asynchronous Receiver Transmitter (UART) where it is divided by 16 to yield a 9600 BAUD clock for serial data transmission.
2. A 614.4 kHz clock which is provided to the ADC0816 Analog-to-Digital Converter.

Thus one of the four timers is available for other uses.

We shall hereafter refer to the two devices as NSC810A #1 and NSC810A #2 respectively. The NSC810A reference manual [Ref. 11] refers to the eight-bit ports by

⁶ The NSC800 includes several instructions not included with the Z-80. Since the Z-80 is the better-known device, we have not used any of the added instructions.

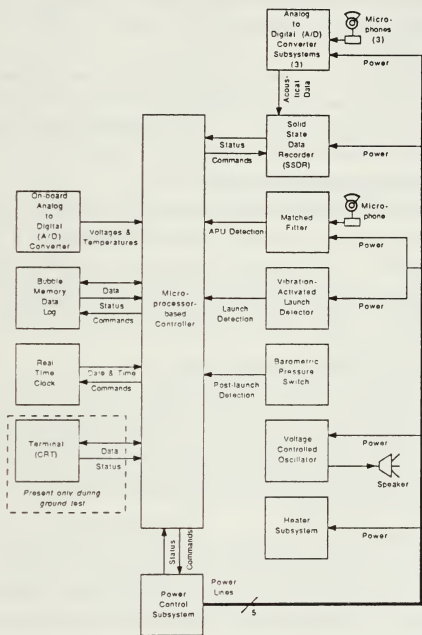


Figure 1. Block diagram of major components of the Vibro-acoustic Experiment.

the letters A and B, and to the six-bit port by the letter C. In the rest of this thesis, when we wish to distinguish between ports on NSC810A #1 and NSC810A #2, we shall append a subscript to the port's letter, *e.g.*, A₁ is NSC810A #1, port A.

NSC810A #1 uses port addresses 0x00 through 0x19.⁷ NSC810A #2 uses port addresses 0x20 through 0x39.⁸

General information on programming the NSC810A can be found in [Ref. 11]. We present the specific manner in which these devices have been programmed for the Vibro-acoustic Experiment in Section A. Major Subroutines and Functions on page 108.

2. On-board Analog-to-digital Converter

An eight-bit, 16-channel, National Semiconductor ADC0816 analog-to-digital (A/D) converter permits the monitoring of voltages and temperatures at various points within the experiment. It can be mounted right on the microprocessor-based controller board. The device is connected starting at input/output space address 0x80. Conversion of an analog input to a digital number is signalled to the control program by the existence of a 1 in bit 3 of port C₂.

3. Bubble Memory Module for the Controller

Devoted to the use of the controller board is a 512 KByte Intel BPK 5V75A Four-Megabit Bubble Memory Prototype Kit. The control program will maintain a log in this memory of all actions it takes during the experiment. The information will include a code signifying the action taken, the time and date of that action, and the current temperature and voltage readings.

After launch, the Vibro-acoustic Experiment is over. The control program then uses the log *solely* for the purpose of recording temperatures and voltages.

Port address 0x40 provides access to the bubble memory. There are 8192 pages of 64 bytes per page. Pages of the bubble memory can be specified at random by num-

⁷ The term "port" is somewhat ambiguous. The NSC810A reference manual [Ref. 11] refers to a collection of pins within an NSC810A integrated circuit as a port. This particular device contains three ports: A, B and C. In common parlance, however, the term port refers to a particular address in the input-output address space (I/O space) of the Z-80. This space spans addresses from 0x00 through 0xff. We might say, for example, that we perform an input operation from port 0x1a. This is equivalent to saying we input a byte (character) from I/O space address 0x1a. We shall seldom attempt explicitly to state which use is intended. The meaning may generally be ascertained from the context.

⁸ See the discussion on hexadecimal notation in the Glossary.

bers from 0 through 8191. Port address 0x41 provides control information for this memory device.

The bubble memory's reset line should be brought low by placing a 0 in bit 5 of port C₂ *before* applying power to or removing power from the bubble memory. It is important to wait at least 50 ms after applying power before attempting to initialize the bubble memory [Ref. 1: Chapter 4, p. 3].

Power can then be applied to the bubble memory by putting a 1 in bit 4 of port C₂. Putting a 0 there removes power.

Details of the operation of this memory and the meaning of the control byte information are in [Ref. 1]. We describe the manner in which we have programmed the bubble memory to support the Vibro-acoustic Experiment in Section A. Major Sub-routines and Functions on page 108.

4. Real Time Clock

A National Semiconductor MM58167A real time clock makes it possible to record in the log of events the dates and times of all actions taken. We also use this device to limit the amount of time the control program waits for various events to occur. If the event does *not* occur for some reason, the control program decides to stop waiting.

For example, once the Auxiliary Power Units (APUs) are detected, there is a window of about seven minutes in length. If launch does not occur within this window, the launch will be scrubbed since the APUs will no longer have sufficient fuel. We can therefore regard the experiment as having been aborted if this amount of time passes without a launch. We use the real-time clock to detect the passage of this period of time.

5. RS-232C Serial Input/Output Port

An RS-232C interface provides communication with a serial device such as a terminal. This makes it feasible to monitor and control the system on the ground. By connecting a terminal to this interface, the user has access to an extensive, menu-driven diagnostic subsystem. (This menu subsystem is dormant if there is no terminal attached.) No intelligence currently is required on the part of the terminal: it is purely a display device.⁹ Port address 0xe0 holds control information to and from the serial device. Port address 0xc0 funnels data either to or from the device. Table 1 on page 14 shows the use of the bits of the control port.

⁹ Mr. David Rigmaiden of the Space Systems Academic Group at the Naval Postgraduate School has proposed encoding the diagnostic messages and using an intelligent terminal to display the corresponding human-readable messages. However, no one has yet done any work on this.

If there is a terminal connected to the RS-232C Serial port at addresses 0xc0 and 0xe0, then bit 3 of port C₁ will be a 1. This permits the control program to distinguish diagnostic operation on the ground (when there *will* be a terminal attached) from actual performance of the experiment (when there *will not* be a terminal attached.) As can be seen in Figure 1 on page 11, the Vibro-acoustic Experiment will not use a terminal when it is in space.

Table 1. ASSIGNMENT OF BITS IN THE RS-232C SERIAL INTERFACE PORT: This port uses address 0x0e for control information and 0x0c for data.

Bit	Direction of Data Flow	Meaning
0	Input	0 if the attached device can accept output information, 1 otherwise.
1	Input	1 if the attached device has data available, 0 otherwise.

B. ADDITIONAL CONTROLLER HARDWARE

The Vibro-acoustic Experiment uses several subsystems which are not a part of the standard controller. Most of these subsystems appear to the right of the microprocessor in the block diagram in Figure 1 on page 11. The only one which does not is the Power Control Subsystem, which is drawn *below* the microprocessor. These subsystems have the following functions:

1. Analog-to-digital Converter Subsystems

Three A/D converters convert the analog acoustical signal detected by a set of three microphones into the digital format required by the Solid State Data Recorder (SSDR). The design and operation of the SSDR is provided in [Ref. 9].

2. Solid State Data Recorder (SSDR)

The SSDR is comparable in function to a conventional reel-to-reel tape recorder. Unlike a standard tape recorder, it is not limited to sequential operation; it can access data randomly. The operation of the SSDR is more fully described in [Ref. 8].

To issue a command to the SSDR, place its code in port A₁ on NSC810A #1, located at I/O space addresses 0x00 through 0x19. The SSDR will place a status code reflecting its operating state in port A₂. Table 4 on page 17 defines the command codes

Table 2. BIT ASSIGNMENTS FOR READING POWER SUBSYSTEM RELAY SETTINGS: The position of relays may be determined by reading port B₂ (on NSC810A #2), which is located at I/O space addresses 0x20 through 0x39.

Bit	Direction of Data Flow	Value	Meaning
0	Not used.		
1	Input	1	The Solid State Data Recorder (SSDR) is on.
		0	The SSDR is off.
2	Input	1	The Voltage Controlled Oscillator (VCO) is off.
		0	It is off.
3	Input	1	The Analog to Digital Conversion (A/D) circuit is on.
		0	It is off.
4	Input	1	The Matched Filter, Vibration-activated Launch detector, and Barometric Pressure Switches are on.
		0	It is off.
5	Input	1	The heater circuit is on.
		0	It is off.

for the SSDR. Table 5 on page 17 defines the status codes the SSDR can return to the control program. The following SSDR commands are of particular note:

Sweep Record 12.5 minutes of pre-determined frequencies emitted by the Voltage Controlled Oscillator (VCO) prior to launch.

Scroll Record up to 55 seconds of ambient noise during the five minutes or so between the time the Auxiliary Power Units (APUs) come on and the time of launch. In this mode, the SSDR will continually re-use the same portion of SSDR memory. There are two sections of memory devoted to this purpose, and use alternates between them. Each can hold up to 55 seconds of ambient noise. When the SSDR is commanded to enter *launch* mode, the memory section currently in use will be filled and then the SSDR will switch over to that section of memory devoted to recording post-launch noise.

Launch The experiment's control program orders the SSDR to enter *launch* mode as soon as the Space Shuttle launches. This mode lasts for two minutes, which is about the time it takes to evacuate the air from inside the shuttle's cargo bay. During this mode, the SSDR records ambient noise.

Table 3. BIT ASSIGNMENTS FOR CONTROLLING POWER SUBSYSTEM RELAYS: Relays may be controlled through port B₁ (on NSC810A #1), which is located at I/O space addresses 0x00 through 0x19.

Bit	Direction of Data Flow	Value	Meaning
0	Output	1	Turn on the relays specified in the other bit positions.
		0	Turn off the relays specified in the other bit positions.
1	Output	1	Operate the Solid State Data Recorder (SSDR).
		0	Do not operate the Solid State Data Recorder (SSDR).
2	Output	1	Operate the Voltage Controlled Oscillator (VCO).
		0	Do not operate the Voltage Controlled Oscillator (VCO).
3	Output	1	Operate the Analog to Digital Conversion (A/D) circuit.
		0	Do not operate the Analog to Digital Conversion (A/D) circuit.
4	Output	1	Operate the Matched Filter (including accelerometer and barometric switch).
		0	Do not operate the Matched Filter.
5	Output	1	Select the heater circuit.
		0	Do not select it.

As can be seen in the flowchart in Figure 22 on page 50, most of the experiment is devoted to the operation of the SSDR, that is, to placing it in the mode appropriate for the current phase of the Space Shuttle's mission.

3. Matched Filter

As mentioned in Section 2. Detection of the Auxiliary Power Units (APUs) on page 4, a matched filter will detect the characteristic 600 Hz signature of the APUs. This device will place a 1 in bit 0 of port C₁ if a detection occurs. Normally it leaves a 0 there. Table 6 on page 18 shows the uses of all the bits of Port C₁. The matched filter is described in Chapter III. THE MATCHED FILTER on page 21.

Table 4. SSDR COMMAND CODES: Commands are issued by writing them to port A_1 on NSC810 #1, located at I/O space addresses 0x00 through 0x19.

Code	Value	Meaning
STANDBY	0x01	Commands the SSDR to cease all operations and await further commands.
SWEEP	0x02	Commands the SSDR to enter <i>sweep</i> mode. Enough memory is available for holding holding 12.5 minutes of noise generated by the VCO.
SCROLL	0x04	Commands the SSDR to enter <i>scroll</i> mode. Enough memory is available for holding 30 seconds of ambient noise.
LAUNCH	0x08	Commands the SSDR to enter <i>launch</i> mode. Enough memory is available for holding two minutes of ambient noise.
RECORD	0x10	Commands the SSDR to start recording noise. This is analogous to the RECORD button on conventional tape recorders.
PLAYBACK	0x20	Commands the SSDR to play recorded data back. This mode is analogous to the PLAY button on conventional tape recorders.

Table 5. SSDR STATUS CODES: Status codes may be obtained by reading them from port A_2 on NSC810 #2, located at I/O space addresses 0x20 through 0x39.

Code	Value	Meaning
OPCOMP	0x40	Shows that the SSDR has completed the last command it received.
NORMOP	0x80	Shows that the SSDR is operating normally.

4. Voltage Controlled Oscillator (VCO)

The purpose of the VCO is to irradiate the shuttle's cargo bay with sound of a predetermined frequency during the *sweep* phase. This is done by applying power to a loudspeaker. The VCO is designed to step up in frequency from 35 Hz through 785 Hz in 1 Hz steps. By recording the echoes, subsequent analysis will permit a comparison of the acoustical response of the pure tone to that of the noise generated during launch.

5. Vibration-activated Launch Detector

This circuit is mounted on the same circuit board as the matched filter. Its purpose is to enable the control program to detect a launch, and so enable it to com-

Table 6. BIT ASSIGNMENTS IN PORT C₁ OF NSC810A #1

Bit	Direction of Data Flow	Value	Meaning
0	Input	1	Detection of the Auxiliary Power Units (APUs) has occurred.
		0	Detection of the Auxiliary Power Units (APUs) has not occurred.
1	Input	1	The Vibration-activated Launch Detector has detected a launch.
		0	The Vibration-activated Launch Detector has not detected a launch.
2	Input	1	One of the barometric pressure switches has detected a launch.
		0	Neither barometric pressure switch has detected a launch.
3	Input	1	No terminal is connected to the port.
		0	A terminal is connected to the RS-232C serial interface port.
4	Output	1	Order the power subsystem to change the states of the relays specified in the command at port B ₁ .
		0	Do not change the states of the relays.
5	Not used.		

mand the Solid State Data Recorder to enter *launch* mode. It will set bit 1 of port C₁ high when it detects a launch (see Table 6 on page 18). Even if the matched filter never detects the APUs, detection of launch will still cause the control program to force the SSDR into *launch* mode.

6. Barometric pressure switches

On the same circuit board as the matched filter there are two barometric pressure switches connected in a redundant, parallel configuration. These switches serve as a backup for the Vibration-activated Launch Detector. Either one of them will place a high voltage in bit 2 of port C₁ (see Table 6) when pressure drops below 27.9 inches of mercury. This pressure corresponds to an altitude between 1500 feet and 2000 feet when the lowest barometric pressures on record at Cape Canaveral are present. In general,

Table 7. BIT ASSIGNMENTS IN PORT C₂ OF NSC810A #2

Bit	Direction of Data Flow	Value	Meaning
0	Output	1	Operate the heater subsystem.
		0	Do not operate the heater subsystem.
1	Not used.		
2	Not used.		
3	Input	1	Analog to digital conversion is complete. This refers to the On-board A D Converter (see Figure 20 on page 48).
		0	Analog to digital conversion is not yet complete.
4	Output	1	Apply power to the bubble memory.
		0	Do not apply power to the bubble memory.
5	Output	1	Do not apply a reset signal to the bubble memory. This is the normal setting.
		0	Apply a reset signal to the bubble memory. This must be done while power is applied to or removed from it. Once the power has been switched on or off, the reset line can be returned to 0.

the corresponding altitude will be somewhat higher than this since barometric pressures will generally not be at their lowest when NASA launches a Space Shuttle.

7. Heater Circuit

The purpose of the heaters is to maintain the temperature of the controller's bubble memory module at or above 10°C during operation, and above -20°C otherwise. To do this, there are heater strips attached to the bubble memory module. To turn on the heaters, the control program places a 1 in bit 0 of port C₂. It puts a 0 there to turn them off. Insufficient power is available to heat all the bubble memories in the Solid State Data Recorder (SSDR). If the contents of the log are saved, however, it should at least be possible to ascertain the cause of the loss of acoustic data in the SSDR.

8. Power Control Subsystem

Three batteries of dry cells, each powering a different bus, provide power to the experiment. Partly in order to conserve power, but also to permit isolation of subsystems if an overheat condition occurs, most subsystems receive power only when neces-

sary. The power subsystem includes a relay for each of these other subsystems. By writing appropriate commands to port B₁ we can turn power to the relays on or off. Table 2 on page 15 shows the uses of the pins of port B₁ for this purpose. By reading a status byte from Port B₁ we can ascertain the position of each relay. Table 3 on page 16 shows the uses of the pins of port B₂ for this purpose.

Let us designate as *relay_i* the relay controlled by pin *i* of port B₁.¹⁰ Valid relays are *relay₁* through *relay₅*.

There is no *relay₀* since bit 0 of port B₁ has a special purpose. If bit 0 of Port B₁ is a 1, then eligible relays will be switched *on*. If bit 0 of Port B₁ is a 0, then eligible relays will be switched *off*. Placing a 1 in bit *i* of Port B₁ makes *relay_i* eligible for switching. Finding a 1 in bit *i* of port B₂ means *relay_i* is on.

Once we have issued a command to alter the position of one of the relays, we place a 1 in bit 4 of port C₁ for 20 ms to permit the command to take effect, then put a 0 in that bit.

¹⁰ Note that pin *i* of port B₁ refers to the same relay as does pin *i* of port B₂.

III. THE MATCHED FILTER

This chapter describes the design of a filter whose purpose is to detect the presence of the 600 Hz tone characteristic of the space shuttle's Auxiliary Power Unit (APU). This circuit has not yet been built and tested, but the most critical sub-circuit, the bandpass filter it uses, has been simulated. The results of the simulation match the predicted performance very closely and are included in this chapter.

The existence of the tone and the equation of a fourth-order elliptical (Cauer) bandpass filter for detecting its presence are documented in Jordan [Ref. 6]. While the thrust of Jordan's work was to develop a digital filter, the implementation described in this thesis uses analog electronics. This implementation has the advantage that it can be constructed expeditiously with readily available components and requires less electrical power. The digital implementation described by Jordan requires special hardware and components. In particular, the Intel 2920 Digital Signal Processor integrated circuit he proposed to use is no longer in production. Jordan does propose an analog implementation in addition to the digital one. It requires six operational amplifiers; the design proposed here requires only four, and so require less power to operate. This is important in this application, since power is limited.

The term *matched filter* ordinarily refers to a particular kind of filter based on autocorrelation. However, the term has come to be applied incorrectly to the bandpass filter used to detect the application of power to the Auxiliary Power Unit. Rather than abandon the term *matched filter*, which has become thoroughly entrenched in the documentation of the Vibro-acoustic experiment, this author will continue to use (misuse) it to denote a narrowband filter whose purpose is to respond to the characteristic 600 Hz tone emitted by the space shuttle's Auxiliary Power Units beginning about five minutes before launch. Figure 2 on page 22 is a block diagram of the author's proposed design for the matched filter.

A. MICROPHONE INPUT STAGE

The microphone input stage is shown in Figure 3 on page 23. It uses a Panasonic WM-063T microphone. A 620 Ω resistor limits the current through the microphone to 8 mA. The output is an AC signal superimposed on a DC bias.

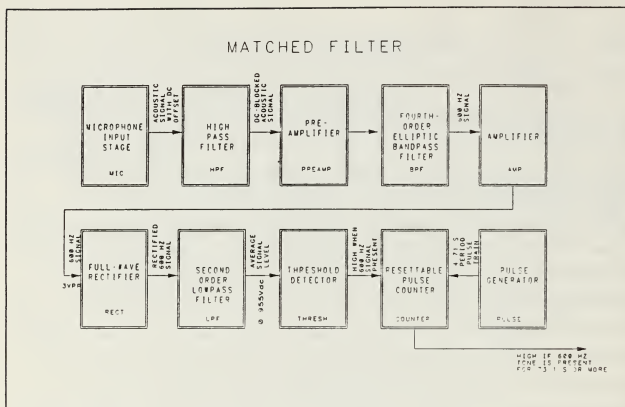


Figure 2. Block diagram of the Matched Filter.

B. HIGH-PASS FILTER

Figure 4 on page 24 shows the high-pass filter which connects the microphone to the pre-amplifier. The purpose of this filter is to eliminate the DC bias from the microphone signal. While simple AC coupling can in principle be provided by a capacitor alone, a resistor to ground must be included to provide a path for DC from the non-inverting lead of the pre-amplifier. Even though the input bias current of the OPA111 operational amplifier used to implement the pre-amplifier is less than 2 pA, if this were neglected, charge would accumulate on the coupling capacitor and the amplifier would saturate.

The cut-off frequency of the high-pass filter was set quite low, at

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(150 \text{ K}\Omega)(150 \text{ nF})} = 7 \text{ Hz.} \quad (2)$$

Since the signal of interest is well above this, at $f = 600 \text{ Hz}$, the filter introduces no significant attenuation or phase shift.

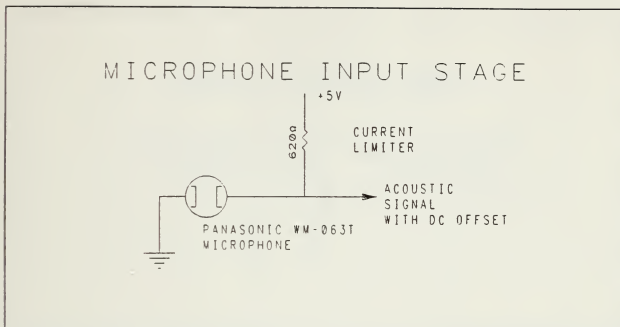


Figure 3. The microphone input stage.

C. PRE-AMPLIFIER

Figure 5 on page 25 shows the pre-amplifier, which boosts the microphone output voltage by a factor of 11 (21 dB) using a non-inverting configuration of the OPA-111 operational amplifier. The OPA-111 has a very low noise of less than $40 \text{ nV}/\sqrt{\text{Hz}}$ at $f = 100 \text{ Hz}$; at higher frequencies it is even lower. Thus the microphone input is boosted to reasonable levels without injecting significant noise into the signal and is buffered prior to the bandpass filter.

D. FOURTH-ORDER, ELLIPTICAL (CAUER), BANDPASS FILTER

Jordan [Ref. 6: p. 45] gives an analog implementation of a fourth-order, elliptical (Cauer), band-pass filter. The design provided below reduces the number of operational amplifiers by two, from six to four.

The coefficients of the necessary transfer function are given in [Ref. 6: p. 36] and are:

$$G(s) = \frac{s^4 + 2.9587 \times 10^7 s^2 + 1.8991 \times 10^{14}}{s^4 + 2.4351 \times 10^7 s^3 + 2.7642 \times 10^7 s^2 + 3.3558 \times 10^9 s + 1.8991 \times 10^{14}}. \quad (3)$$

The author used a computer program to find the roots of this transfer function, which can be rewritten as follows:

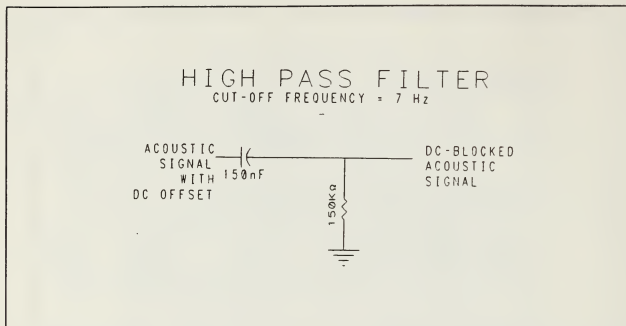


Figure 4. High-pass filter.

$$G(s) = \frac{(s + j4.49 \times 10^3)(s - j4.49 \times 10^3)(s + j3.07 \times 10^3)(s - j3.07 \times 10^3)}{(s + 62 + j3.84 \times 10^3)(s + 62 - j3.84 \times 10^3)(s + 58.8 + j3.59 \times 10^3)(s + 58.8 - j3.59 \times 10^3)} \quad (4)$$

By multiplying together the terms containing complex conjugates of each other, we obtain the biquadratic representation of this function.

$$G(s) = \left[\frac{s^2 + (3.068 \times 10^3)^2}{s^2 + \left(\frac{3.586 \times 10^3}{30.5} \right)s + (3.586 \times 10^3)^2} \right] \left[\frac{s^2 + (4.491 \times 10^3)^2}{s^2 + \left(\frac{3.843 \times 10^3}{31.0} \right)s + (3.843 \times 10^3)^2} \right] \quad (5)$$

Each of the factors in this expression has been written in the form

$$F(s) = \frac{s^2 + \omega_z^2}{s^2 + \left(\frac{\omega_p}{Q_p} \right)s + \omega_p^2} \quad (6)$$

This is the equation of a notch filter, given by Ghausi [Ref. 12: p. 16]. If $\omega_z = \omega_p$, then the notch filter is symmetric, that is, the attenuation curve to the left and right of the notch frequency is symmetric about that frequency when the transfer function is plotted on a logarithmic frequency scale. The first factor in equation (5) has $\omega_z < \omega_p$. Consequently, this factor represents a *high-pass* notch filter. The magnitude of $G(s)$ rises once ω in $s = j\omega$ exceeds ω_z ; it levels off once ω exceeds ω_p . By contrast, the second factor in

PRE-AMPLIFIER

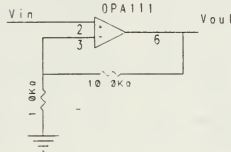


Figure 5. Pre-amplifier.

equation (5) has $\omega_s > \omega_p$. Therefore, this factor represents a *low-pass* notch filter. The magnitude of $G(s)$ drops once ω in $s = j\omega$ exceeds ω_p ; it levels off once ω exceeds ω_s . A low-pass notch filter and a high-pass notch filter placed in cascade form a bandpass filter if suitable choices for ω_p and ω_s are made in each case.

It can be difficult to implement cascaded filters successfully. However, the cascade filter is very attractive due to its simplicity, and for this reason we have employed it here. The design presented has been simulated and so we believe it would be quite straightforward to implement it in hardware.

The three forms of notch filter and the bandpass filter formed by cascading a low-pass and a high-pass notch filter are shown in Figure 9 on page 29. Three of these curves were calculated by computer from the factors in the transfer function given in equation (5). The symmetrical notch filter transfer function plotted in the figure is an example of what results from equation (6) when $\omega_p = \omega_s$. It, too, was calculated by computer from the transfer function. When the high-pass notch filter is multiplied by (put in cascade with) the low-pass notch filter, the bandpass filter shown in the figure results. By using asymmetrical notch filters, as opposed to symmetrical ones, we can obtain high gain in the passband. In this region both notch filters have high gain and so reinforce one another.

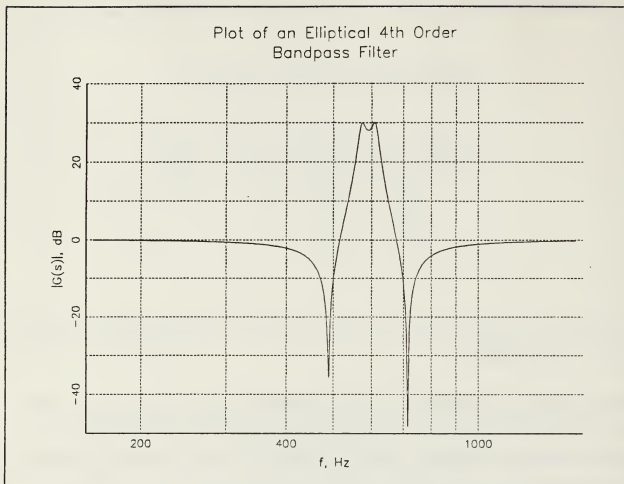


Figure 6. Magnitude of the transfer function of the elliptical bandpass filter.

The transfer function for equation (5) is plotted separately in Figure 6 on page 26; this plot, too, was generated by computer. The advantage to writing the equation for the elliptical band-pass filter in the form of equation (5) is that it is a comparatively simple matter to implement biquadratic filter sections using operational amplifiers; by cascading these sections, the entire transfer function can be implemented. Again, it can be difficult to implement this scheme.

Figure 7 on page 27 shows a schematic for a generalized biquadratic filter using two operational amplifiers. The blocks labeled with the letter "Y" represent admittances. The design equations for these two filter sections are derived in APPENDIX A. Derivation of Design Equations for the Matched Filter on page 71. For the high-pass notch filter, they are

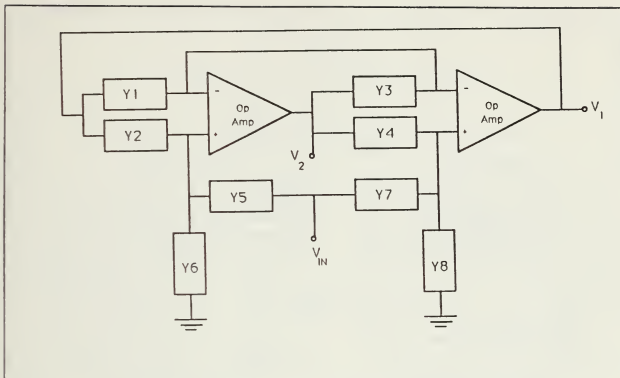


Figure 7. A generalized biquadratic filter using two operational amplifiers.

$$Y_1 = C_a \Leftrightarrow Z_1 = \frac{1}{C_a} \quad (7)$$

$$Y_2 = sC_a \Leftrightarrow Z_2 = \frac{1}{sC_a} \quad (8)$$

$$Y_3 = C_b \Leftrightarrow Z_3 = \frac{1}{C_b} \quad (9)$$

$$Y_4 = Y_5 = \frac{1}{R} \Leftrightarrow Z_4 = Z_5 = R \quad (10)$$

$$Y_6 = 0 \Leftrightarrow Z_6 = \infty \quad (11)$$

$$Y_7 = sC_b \Leftrightarrow Z_7 = \frac{1}{sC_b} \quad (12)$$

$$Y_8 = \frac{1}{Q_p R} \Leftrightarrow Z_8 = Q_p R \quad (13)$$

FOURTH-ORDER ELLIPTIC BANDPASS FILTER

GAIN = 30 dB AT $f = 600$ Hz
 ATTENUATION = 0 dB AT $f < 500$ Hz AND $f > 700$ Hz
 3 dB BANDWIDTH = 50 Hz

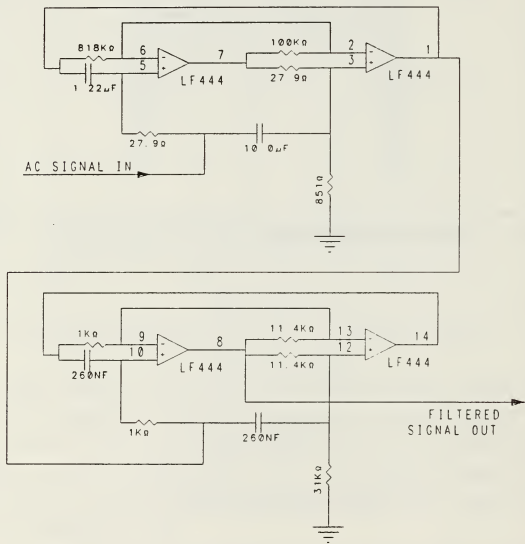


Figure 8. A fourth-order, elliptic bandpass filter with $Q = 12$: It provides 30 dB of attenuation outside the passband.

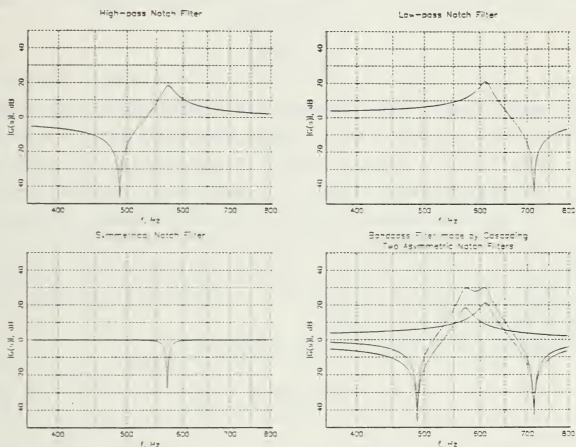


Figure 9. Notch filters: Symmetric, low-pass, and high-pass notch filters; and a bandpass filter formed from a low-pass and a high-pass notch filter in cascade.

$$\frac{C_b}{C_a} = Q_p \left[1 - \left(\frac{\omega_z}{\omega_p} \right)^2 \right] \quad (14)$$

$$R = \frac{1}{\omega_p C} . \quad (15)$$

For the high-pass notch filter, $\omega_z = 3.068 \times 10^3$, $\omega_p = 3.586 \times 10^3$, and $Q_p = 30.5$. Hence

$$\frac{C_b}{C_a} = 8.176. \quad (16)$$

If we make the arbitrary choice $C_b = 10 \mu\text{F}$, then we get $C_a = 1.22 \mu\text{F}$, $R = 27.9 \Omega$, $Z_3 = R_3 = 100 \text{ K}\Omega$, $Z_1 = R_1 = 818 \text{ K}\Omega$, and $Q_p R = 851 \Omega$. Note that Z_1 is a resistor whose magnitude in ohms is the reciprocal of the magnitude of C_a in farads. Similarly, Z_3 is a resistor whose magnitude in ohms is the reciprocal of the magnitude of C_b in farads. The apparently arbitrary choice for C_b was actually not random. This circuit must be made with components whose stability is high to minimize changes in performance due to changes in temperature. Capacitors with low temperature coefficients are available in polystyrene up to values of $10 \mu\text{F}$. Using this value for C_b allows R_1 not to be too big, and R not to be too small.

For the low-pass notch filter, the design equations are

$$Y_1 = Y_5 = \frac{1}{R_b} \Leftrightarrow Z_1 = Z_5 = R_b \quad (17)$$

$$Y_2 = Y_7 = sC \Leftrightarrow Z_2 = Z_7 = \frac{1}{sC} \quad (18)$$

$$Y_3 = Y_4 = \frac{1}{R_a} \Leftrightarrow Z_3 = Z_4 = R_a \quad (19)$$

$$Y_6 = 0 \Leftrightarrow Z_6 = \infty \quad (20)$$

$$Y_8 = \frac{1}{Q_p R_b} \Leftrightarrow Z_8 = Q_p R_b \quad (21)$$

$$\frac{R_a}{R_b} = Q_p \left[\left(\frac{\omega_z}{\omega_p} \right)^2 - 1 \right] \quad (22)$$

$$C = \frac{1}{R_b \omega_p} \quad (23)$$

For the low-pass notch filter, $\omega_s = 4.491 \times 10^3$ rad/s, $\omega_p = 3.843 \times 10^3$ rad/s, and $Q_p = 31.0$. So

$$\frac{R_a}{R_b} = 11.35. \quad (24)$$

If we arbitrarily pick $R_b = 1$ K Ω , then $R_s = 11.4$ K Ω , $C = 260$ nF, and $Q_p R_b = 31$ K Ω . Figure 8 on page 28 shows the complete bandpass filter. We have used the LF444 Quad Low Power JFET Input Operational Amplifier. It has an extremely low input bias current of 50 pA at most, and only 35 nV/ $\sqrt{\text{Hz}}$ noise voltage.

We simulated the frequency response of this filter using Micro-Cap III [Ref. 13].¹¹ We found that it performed almost exactly as predicted. Figure 10 on page 32 is a plot generated by Micro-Cap III from its simulation. By comparing this plot with that generated from the transfer function in Figure 6 on page 26, we see that the only departure from the predicted performance is a slight asymmetry in the ripple in the passband. Since we are concerned only with detection of the Auxiliary Power Units' acoustic signature, and not with faithful reproduction, this is not a matter for concern. The center of the passband and the location of the upper and lower notch frequencies are at the predicted frequencies. The gain in the passband also is as predicted. The simulation results are strong evidence of the correctness of the analysis and the feasibility of the design.

The operational amplifier in the Pre-amplifier is an OPA111. Its output impedance is 100 Ω . The input impedance of the bandpass filter is well above 10 k Ω throughout the passband. The bandpass filter therefore does not provide a significant load on the Pre-amplifier, and so the simulation results can be considered to be quite accurate, even though they were produced with an assumption of zero output impedance from the Pre-amplifier.

¹¹ In the simulation, we used two LF442 operational amplifier packages instead of a single LF444 operational amplifier package. These two packages provide operational amplifiers with identical electrical characteristics, which justifies the substitution made.

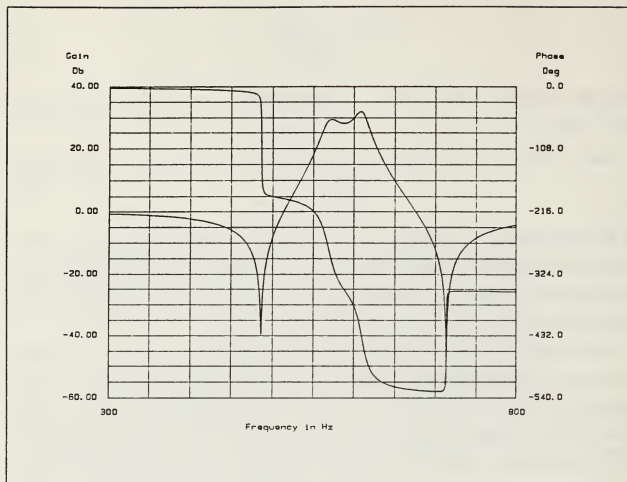


Figure 10. Frequency response of the simulated bandpass filter: This plot was obtained using Micro-Cap III [Ref. 13]. The phase response also is shown. It is the curve with the staircase-like appearance. The gain response is nearly identical with that generated by computer and shown in Figure 6 on page 26.

E. ADJUSTABLE GAIN

Figure 11 on page 33 shows how a single LF444 operational amplifier is configured as a non-inverting amplifier of variable voltage gain up to 28 (29 dB). The gain is to be adjusted so that the strongest output signal has 3 V peak-to-peak. This maximum signal is that which exists when the Auxiliary Power Unit outputs its characteristic tone.

F. FULL-WAVE RECTIFIER

Figure 12 on page 34 shows the design of a full-wave rectifier. It converts the 600 Hz tone admitted by the band-pass filter into a fluctuating direct current signal. This

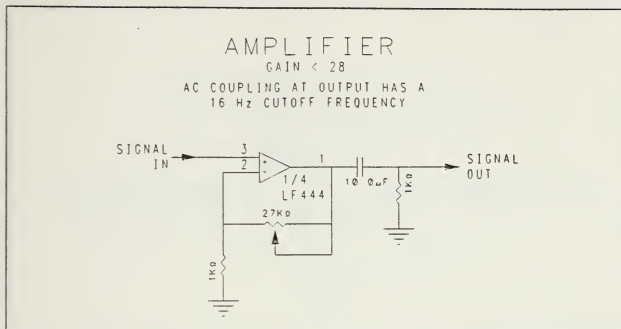


Figure 11. Amplifier providing a variable voltage gain up to $28 = 28.9$ dB.

circuit is modified from an absolute value circuit provided by Jung [Ref 14: pp. 236-237]. It operates as follows:

The inverting terminal of both operational amplifiers is at virtual ground. Therefore, on the positive cycle of the incoming signal, current passes through resistor R_1 to the inverting input of the first operational amplifier. This current is unable to enter the operational amplifier because of its extremely large input impedance; nor can it pass through diode D_2 , since that diode will only pass current in the other direction. Consequently, it passes through resistor R_2 . At point a , the voltage is the negative of the input voltage. The same amount of current flows through resistor R_3 . Resistor R_4 has only half the resistance of R_3 , so it draws twice the current that resistor R_3 can supply. The balance comes through resistor R_5 . This causes the output of the second operational amplifier to match that of the input to the circuit. So during the positive cycle of the input, the input voltage is duplicated at the output.

On the negative cycle, the two diodes serve to keep the voltage at point a at ground potential. This eliminates all current through resistors R_2 and R_4 . The effect is the same as if the first operational amplifier were removed entirely. The second operational amplifier is then in the usual configuration for inverting. The inverse of the negative input signal is, of course, a positive signal.

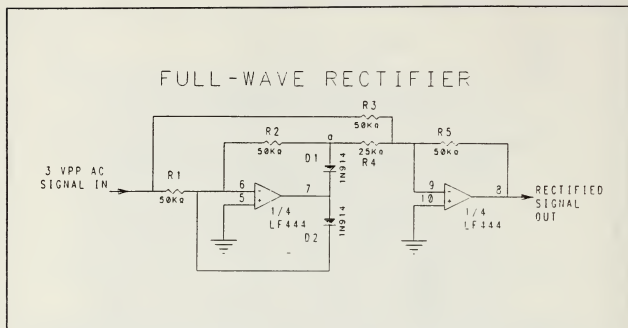


Figure 12. Full-wave rectifier.

In summary, whether the input is positive or negative, the output is the absolute value of the input.

G. LOW-PASS FILTER

The signal out of the rectifier has a fundamental frequency of $2 \times 600 \text{ Hz} = 1200 \text{ Hz}$ and this is superimposed on a DC voltage derived as follows:

$$\begin{aligned} v(t) &= V' \sin(2\pi f t) \\ &= V' \sin(\omega t) \end{aligned} \quad (25)$$

Generalized Second-Order Low-pass Filter Using One OPAMP

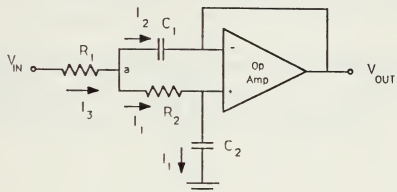


Figure 13. A general second-order, single operational amplifier, low-pass filter.

SECOND-ORDER LOW-PASS FILTER

CUTOFF FREQUENCY = 5 Hz

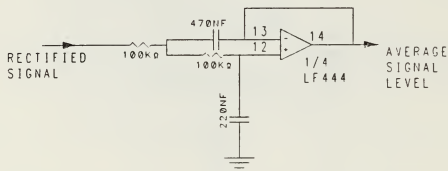


Figure 14. Second-order, low-pass filter.

$$\begin{aligned}
 V_{AV} &= \frac{1}{\left(\frac{T}{2}\right)} \int_0^{\frac{T}{2}} v(s) dt \\
 &= \frac{2}{T} \int_0^{\frac{T}{2}} V \sin(\omega t) dt \\
 &= \frac{2V}{T} \left[-\frac{\cos(\omega t)}{\omega} \Big|_0^{\frac{T}{2}} \right] \\
 &= \frac{2V}{T} \left[-\frac{\cos\left(\frac{2\pi f T}{2}\right)}{\omega} + \frac{1}{\omega} \right] \\
 &= \frac{2V}{T} \left[-\left(-\frac{1}{\omega}\right) + \frac{1}{\omega} \right] \\
 &= \frac{4V}{T\omega} \\
 &= \frac{4V}{12\pi f} \\
 &= \frac{4V}{2\pi} \\
 &= \frac{2V}{\pi} .
 \end{aligned} \tag{26}$$

The signal into the filter has a peak amplitude of 1.5 V. Application of this formula gives $V_{AV} = 0.955$ V. This is the amplitude of the strongest signal we expect to receive from the Auxiliary Power Unit.

The low-pass filter which follows the rectifier is designed to have a cut-off frequency of $f_c = 5$ Hz. This frequency is well below the fundamental frequency of 1200 Hz passed by the rectifier. As a consequence, only the average signal V_{AV} we have just derived will be present at the output.

The circuit is based on the general circuit shown in Figure 13 on page 35. The design equations for this circuit are derived in APPENDIX A. Derivation of Design Equations for the Matched Filter on page 72 and are reproduced here.

$$C_1 = 4C_2Q_p^2 \quad (27)$$

$$R_1 = R_2 = R = \frac{1}{\omega_p \sqrt{C_1 C_2}} \quad (28)$$

In this application, we are not concerned with the phase of the signal. Therefore it is reasonable to seek a maximally flat transfer function. To do this, we implement a second-order Butterworth filter, for which

$$Q_p = \frac{1}{\sqrt{2}} = 0.707.$$

Given our nominal cut-off frequency $f_c = 5$ Hz, we arbitrarily choose $C_2 = 220$ nF. We get $C_1 = 440$ nF, and $R = 102.3$ k Ω . Since the cut-off frequency is not critical, we can pick the more convenient component values $C_1 = 470$ nF and $R = 100$ k Ω . The resultant cut-off frequency is $f_c = 4.9$ Hz and $Q_b = 0.731$. Since the purpose of this low-pass filter is to find the average of the rectified 600 Hz tone, this deviation from the design parameters is quite acceptable as the cut-off frequency still is well below the fundamental frequency of 1200 Hz created by full-wave rectification of the 600 Hz tone.

The chief benefit provided by this filter is a roll-off of 40 dB/decade when $f > f_c = 4.9$ Hz. This amounts to 96 dB attenuation when $f = 1200$ Hz, which is ample to suppress the AC component in the signal out of the full-wave rectifier. Figure 14 on page 36 shows the component choices and circuit for the second-order low-pass filter.

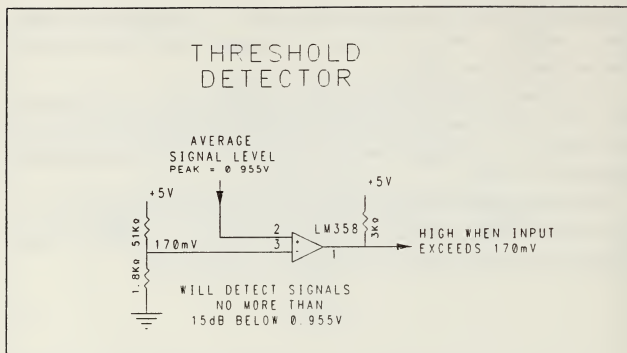


Figure 15. Threshold detector.

H. THRESHOLD DETECTOR

Figure 15 on page 38 shows the design of a threshold detector. There is some evidence that other sources of 600 Hz signals that might be present simultaneously will be 15 dB below this. The voltage which is 15 dB below 0.955 V is 0.170 V. Any signal which exceeds the 0.170 V threshold just derived should cause the threshold detector to signal that a sufficiently strong 600 Hz signal is present. Presumably this signal is from the Auxiliary Power Unit. The threshold detector uses an LM358 operational amplifier. This device requires only a single power supply and it tends not to "latch up" when configured as a comparator. Its output goes high (to 5 V) whenever the threshold is exceeded. Otherwise, its output is held low (at ground).

I. RESETTABLE PULSE COUNTER

Figure 16 on page 39 shows the Resettable Pulse Counter. Its purpose is to signal the presence of a 600 Hz signal from the Auxiliary Power Unit if it has been continuously present for 73.1s. Spurious signals with a component at $f = 600$ Hz may be present intermittently. We do not expect them to be continuously present at levels more than 15 dB below that of the strongest signal expected from the Auxiliary Power Unit.

RESETTABLE PULSE COUNTER

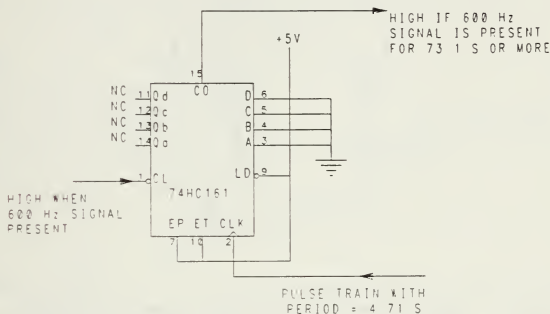


Figure 16. Resetable Pulse Counter: This circuit decides that the APU's are on if it gets a signal from the threshold detector for 73.1 s.

Consequently, the Resetable Pulse Counter has the effect of eliminating false triggering due to these spurious signals.

Whenever the threshold detector indicates the presence of the 600 Hz tone characteristic of the Auxiliary Power Unit, it produces a high output. This signal is applied to the LOAD(L) input of the Resetable Pulse Counter. This permits the counter to begin marking off the pulses which arrive from the Pulse Generator, described below. Because the pre-load inputs *A* through *D* all are connected to ground, the counter will count from zero to 15, at which point its CO output will go high. Thus, the counter will permit sixteen pulses to arrive from the Pulse Generator before it goes high. Since the period of these pulses is $T = 4.57$ s, the output will go high if 16×4.57 s = 73.1 s elapses.

Configuration of an LM555 Timer for Astable Operation

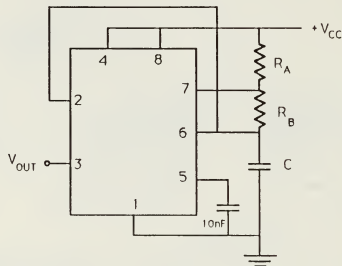


Figure 17. Astable operation of the LM555 Timer to generate a pulse train.

J. PULSE GENERATOR

This module is based on the LM555 Timer integrated circuit. The data sheet for this circuit provides equations to permit choosing component values to provide the desired period and duty cycle. The duty cycle is not critical to this application. Figure 17 on page 40 shows the general configuration for astable operation, which is the mode of operation which produces a periodic signal. The design equations are:

$$t_{charge} = 0.693(R_A + R_B)C \quad (30)$$

$$t_{discharge} = 0.693R_B C. \quad (31)$$

Thus the total period

$$T = t_{charge} + t_{discharge} = 0.693(R_A + 2R_B)C. \quad (32)$$

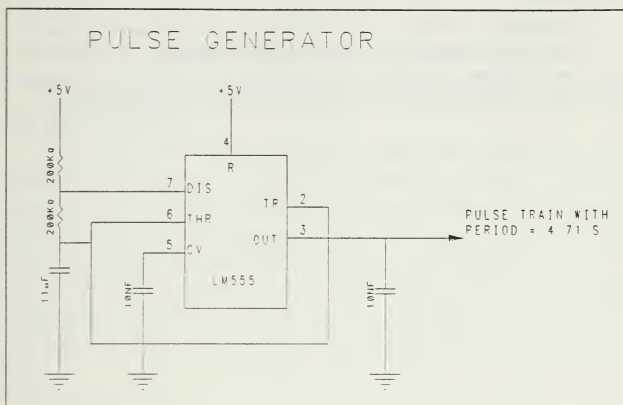


Figure 18. Pulse Generator.

Solving for C we get

$$C = \frac{T}{0.693(R_A + 2R_B)} = 11 \mu\text{F} \quad (33)$$

and the duty cycle

$$D = \frac{R_B}{R_A + 2R_B} \quad (34)$$

Picking $R_A = R_B = 200 \text{ K}\Omega$ provides a duty cycle $D = 0.333$, which is perfectly acceptable for this non-critical parameter.

Figure 18 shows a Pulse Generator based on this configuration. This circuit produces a regular stream of square pulses of period $T = 4.57 \text{ s}$. These are used by the Resettable Pulse Counter to measure the amount of time when a reasonably strong 600 Hz signal is present.

K. SUMMARY

This completes the description of the somewhat inappropriately named matched filter. Simulation of the bandpass filter used in the matched filter has shown that that part of the design is correct and feasible. The implementation and testing of the bandpass filter and the other components of the entire circuit remain to be done in the future.

IV. DESIGN OF THE CONTROL SOFTWARE

In describing the software which operates the controller hardware, we shall adopt the following conventions.

We will show variable names in bold, *e.g.*, **variable**; function names in bold with (possibly empty) parentheses at the end,¹² *e.g.*, **function()**, and constants in uppercase, *e.g.*, **CONSTANT**.¹³ We shall also use **bold** for the names of regions, described below in Section A. Memory Map. Development of the software for the Vibro-acoustic Experiment was done under the Microsoft Disk Operating System (MS DOS). Figure 35 on page 89 shows how we arranged the hierarchy of files containing the source code, object code, header files, *etc.* See APPENDIX D. HIERARCHICAL ORGANIZATION OF SOFTWARE FILES on page 88 for a more complete discussion of this organization.

A. MEMORY MAP

Figure 19 on page 44 shows the addresses of the ROM and RAM in the computer. Our NSC800-based controller provides for up to eight EPROMs and RAMs in any combination, each holding 8 KBytes. The wiring of the printed circuit board permits placing a RAM chip in any of the addresses evenly divisible by 0x2000 (*e.g.*, 0x0000, 0x2000, 0x4000, *etc.*) The addition of a jumper wire permits the RAM chip to be replaced by an EPROM.

The NSC800 uses the same architecture as the Z-80 [Ref. 10]. Because the Z-80 architecture causes execution to begin at address 0x0000 whenever power is applied, it is necessary to install an EPROM at location 0x0000. It was therefore convenient for us to put all EPROMs at the low end of memory, and all RAMs at the high end. APPENDIX C. HOW THE UNIWARE SOFTWARE USES THE COMPUTER MEMORY on page 86 explains the way in which the Uniware C Compiler employs the memory.

¹² According to custom in the C programming language.

¹³ In C, constants are declared using the **#define** directive. These are stored in various header files such as **vibro.h**.

Memory Address	Memory Type	Memory Usage
0x0000	ROM	reset
0x2000	ROM	code
0x4000	ROM	const
0x6000	ROM	string
		data
0x8000	NOT IN USE	
0xa000		
0xc000		
0xe000	RAM	data & ram mbrkram stack

Figure 19. Memory map of the computer: This figure shows the locations of ROM, RAM, and the eight software regions. The ROM and RAM addresses are specified by the hardware design. The addresses of the regions are specified by the linker.

B. OPERATION OF THE VIBRO-ACOUSTIC EXPERIMENT

1. Menu-driven Diagnostic Program

Some years ago the film *Alien* was produced. As a part of the publicity campaign attending its release was the slogan, "In space, no one can hear you scream." Similarly, when the Vibro-acoustic Experiment is performed in the Space Shuttle, there will be no one to hear it scream, that is, to monitor the progress of the experiment.

This is quite different from the situation on the ground, where generally there *is* a monitor attached, and there *is* someone monitoring execution of the program. Furthermore, there is a need on the ground to test components of the experimental package without running the experiment from start to finish. For example, we have found it helpful to be able to operate the bubble memory module attached to the controller hardware in a manual mode. By this means, we have debugged the software and ensured that it *can* operate the bubble memory successfully before attempting to use it in our application.

An obvious way to allow software to be tested on the ground but used to run the experiment in space would be to compile a different program for each purpose. This is, to put it mildly, a very inconvenient approach. Not only must two distinct programs be managed, but assurance that the diagnostic version works gives little assurance that the operational one will work. We have elected to have a single program, usable under all circumstances. This requires that the program be able to recognize that someone is monitoring it. To do this, it simply checks bit 3 of port C_1 to see if a terminal is connected to the RS-232C serial interface. (See Table 6 on page 18.) If there is *no* terminal attached, the program assumes that there is no one *monitoring* execution, *i.e.*, the experimental package might very well be in space, and so the experiment should be operated. If there *is* a terminal attached, then the package cannot possibly be in space, since we don't intend to include a monitor in the Space Shuttle. In this case, the control program does not operate the experiment. Instead, it presents to the operator a series of menus of choices from which the operator can choose which part of the system to exercise.

The menu subsystem provides the following capabilities:

1. **Software reset.** This has an effect similar to that caused by removing power and then applying it again.
2. **Real-time clock control.** The user can set the date and time, read them, or test a time-out feature used to synchronize events during the experiment.

3. **Power subsystem control.** Individual subsystems can be powered up or down under the user's control.
4. **Bubble memory control.** The bubble memory used for storing a log of events performed during the experiment can be powered up or down, initialized, or tested under the user's control. These tests are at a very low level. Data stored in the bubble memory consists of character strings, and these are unformatted. The data are not treated as formatted log entries.¹⁴
5. **Analog-to-Digital (A/D) converter control.** Any of the temperatures and voltages accessible through a channel of the on-board A/D converter can be read under the user's control.
6. **Running the experiment.** This causes the Vibro-acoustic experiment to be performed. The only difference between operating the experiment under menu control and operating it with no terminal attached is that *with* a terminal, a large amount of diagnostic information is displayed during execution. *Without* a terminal attached, this information is lost. The advantage to this approach is that if the experiment works on the ground, we are assured it will work in space, since essentially the same code is executed in both cases.
7. **Perform port input and output.** This is a very low-level test. Characters of data can be written to or read from a port at any address, one at a time. This is helpful in debugging the software.
8. **Display contents of the controller's memory.** This, too, is a very low-level routine useful only for debugging.
9. **Examine or change bubble memory.** These routines permit the formatted contents of the bubble memory log to be displayed in a readable manner on the terminal. In addition to allowing debugging to be done, this operation permits the experiment's operation to be tailored in advance.

Each of these menu items leads to a further menu of functions to permit the operator to test all subsystems of the experiment. These routines are discussed in detail in the software description contained in Section A. Major Subroutines and Functions on page 108.

2. Performing the Experiment

This section describes in detail the steps of the experiment. These steps are illustrated in the flowcharts contained in the following pages.

a. Microprocessor Control Program

Flowchart 0 in Figure 20 on page 48 shows the overall structure of the control program. The program begins executing when power is first applied to the system. After initializing the hardware for proper operation, it checks to see whether or not there is a terminal attached. If not, it proceeds on the footing that it should run the

¹⁴ There are additional bubble memories within the Solid State Data Recorder (SSDR) which are *not* tested by these routines.

experiment, bypassing all the menu routines. If, on the other hand, a terminal is connected, then the program deduces that the experiment is not being run in space, where no terminal will be available, and so it enters the menu subsystem. The experiment is not performed unless the user specifically requests this later.

b. Initialize Hardware

Flowchart 1 in Figure 21 on page 49 is a more detailed look at block 1 of Flowchart 0 in Figure 20 on page 48. The first initialization task to be performed is to let the programmable input/output devices know which data lines are for input and which are for output. There are two clocks which also must be initialized. One of these provides a clock signal for serial communications at 9600 baud. The other provides a clock for conversion of data from analog to digital form.

c. Run the Vibro-acoustic Experiment

Flowchart 2 is shown in Figure 22 on page 50. It is a more detailed look at block 2 of Flowchart 0 in Figure 20 on page 48.

One of its first tasks is to initialize certain variables in the software. It then ascertains (by consulting a record in the bubble memory) whether the full experiment or the abridged version is to be performed. The full experiment consists of the *sweep*, *scroll*, *launch*, and *post-launch* phases already described in Chapter 1. INTRODUCTION on page 1. The *scroll* phase is omitted if the Auxiliary Power Units (APUs) are not detected before the launch was detected.

In the abridged experiment, the program initially checks to see if the barometric switches have been triggered, which they would have been were the Space Shuttle already in space. This check is done to avoid entering *record* phase a second time when the space shuttle is already aloft. Such a situation might arise after a power fault during lift-off; to recommence *record* mode would erase the acoustic data recorded during the launch.

The next decision to be made is whether or not to enter *record* mode. Conceivably, the Auxiliary Power Units could be detected and the *record* phase entered at some point, but the launch might then be scrubbed. If power were not removed from the experiment, then the control flow would permit *record* mode to be commenced anew. Why not just start *record* mode again? Operating the Solid State Data Recorder with its bubble memories consumes considerable power. We cannot afford to waste that power by, in effect, continuously operating the recorder unnecessarily. The decision not to let this mode be begun again until at least 12 hours after the last time will prevent this from happening. Also, if a power fault occurred after the *record* phase began, we would

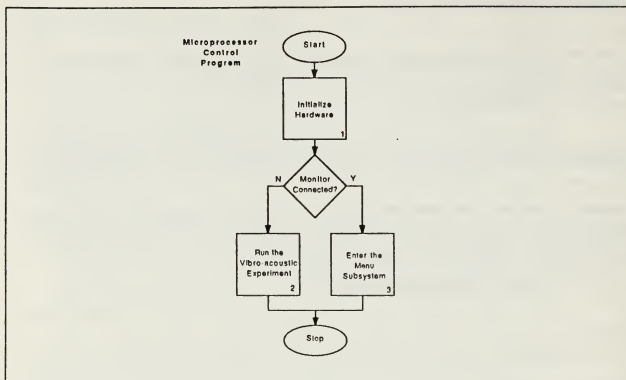


Figure 20. Flowchart 0: This is the highest level of flowchart in the hierarchy. Processing begins here when the controller first receives power.

prefer to avoid interfering with the Solid State Data Recorder (SSDR), which might still be operating successfully in *record* mode. This safeguard will ensure that such interference does not occur. If a mission is scrubbed, it would not be rescheduled for at least 24 hours. The 12 hour wait is long enough to avoid interfering unduly with the SSDR and to preclude wasting power and is short enough to permit correct operation when the launch is rescheduled.

Once either the Auxiliary Power Units or any launch indication are detected, *record* mode can be entered. Normally, upon completion, we expect to be in space. In this case, control will be passed to the *post-launch* phase of the mission. Otherwise, the mission must have been aborted and the 12 hour wait begins.

d. Initialize Software

Flowchart 2.1 in Figure 23 on page 51 is a more detailed look at block 2.1 of Flowchart 2 in Figure 22 on page 50. Initializing the software entails discovering what the current status of the experiment is. For example, this might be the first time power has been applied, in which case the *sweep* phase has not been performed yet, and

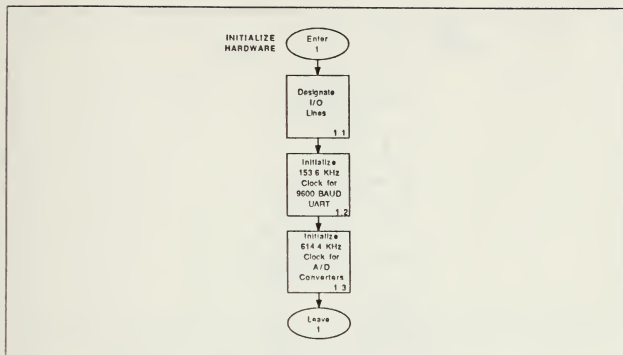


Figure 21. Flowchart 1: The flowchart shows the initialization of hardware and software at the very beginning of program execution.

the launch has not taken place. Or perhaps the *sweep* was performed previously, but the Space Shuttle still is on the ground. This information was stored previously in the bubble memory log, and it must be retrieved before the controller can know what it should do.

The controller also turns off the Voltage Controlled Oscillator for safety reasons. Because the speaker connected to it emits such a loud tone, it would be dangerous to allow it to operate until the controller can first see whether it has been begun once before. If NASA eventually agrees to permit the *sweep* phase to be performed, they will almost certainly afford us only one opportunity to perform it. If it does not complete successfully, there is no second chance. The heater subsystem also is deactivated as a power-saving measure until the controller can find out exactly what to do.

e. Do Sweep

Flowchart 2.2 in Figure 24 on page 52 is a more detailed look at block 2.2 of Flowchart 2 in Figure 22 on page 50. If the *sweep* phase ever was started before, or if the launch was performed previously, the *sweep* phase is skipped. Otherwise the controller notes in the bubble memory log that it has now started the *sweep* phase, which

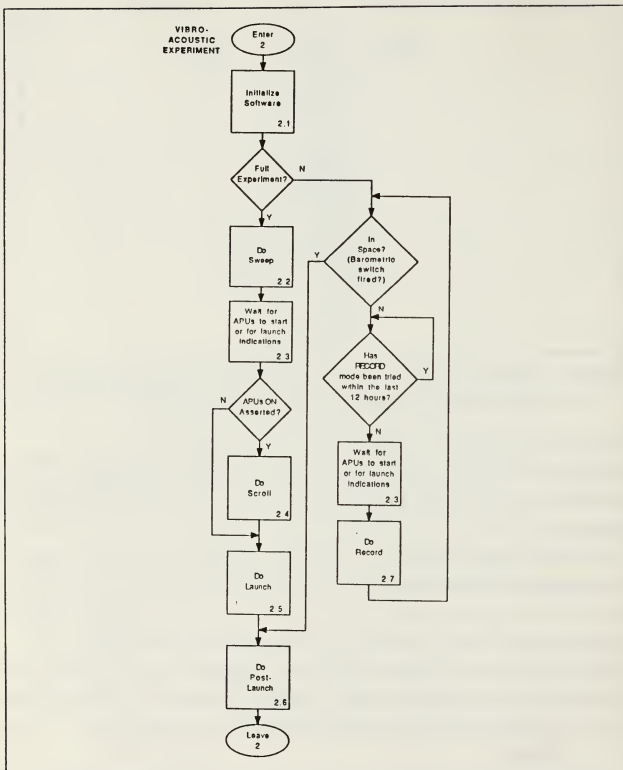


Figure 22. Flowchart 2: This flowchart shows the steps entailed in both the full and the abridged versions of the Vibro-acoustic Experiment.

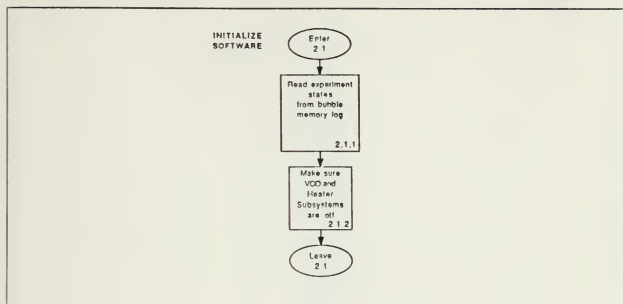


Figure 23. Flowchart 2.1: This flowchart shows the steps entailed in initializing the software when the experiment is performed.

will ensure that it never tries to restart it. The controller now causes echoes of known frequencies to be recorded.

The controller expects to be informed of the completion of the *sweep* phase. However, a 13 minute timeout is initiated to make sure that the controller does not wait forever for this information.

f. Start Recording Response at Known Frequencies

Flowchart 2.2.2 in Figure 25 on page 53 is a more detailed look at block 2.2.2 of Flowchart 2.2 in Figure 24 on page 52. It shows the steps entailed in initiating the *sweep* phase. The Analog to Digital Converter Subsystem must be turned on first, since the converters power the microphones which receive the acoustic signal. The Voltage Controlled Oscillator (VCO) can then be started, followed by the Solid State Data Recorder (SSDR). Starting the SSDR requires first applying power to it and then commanding it to enter *sweep* mode.

g. Stop Recording Response at Known Frequencies

Flowchart 2.2.4 in Figure 26 on page 54 is a more detailed look at block 2.2.4 of Flowchart 2.2 in Figure 24 on page 52. It shows the steps entailed in terminating the *sweep* phase. These steps are to remove power from three subsystems: the Voltage Controlled Oscillator (VCO), the Solid State Data Recorder (SSDR) and the Analog to Digital (A/D) Converter.

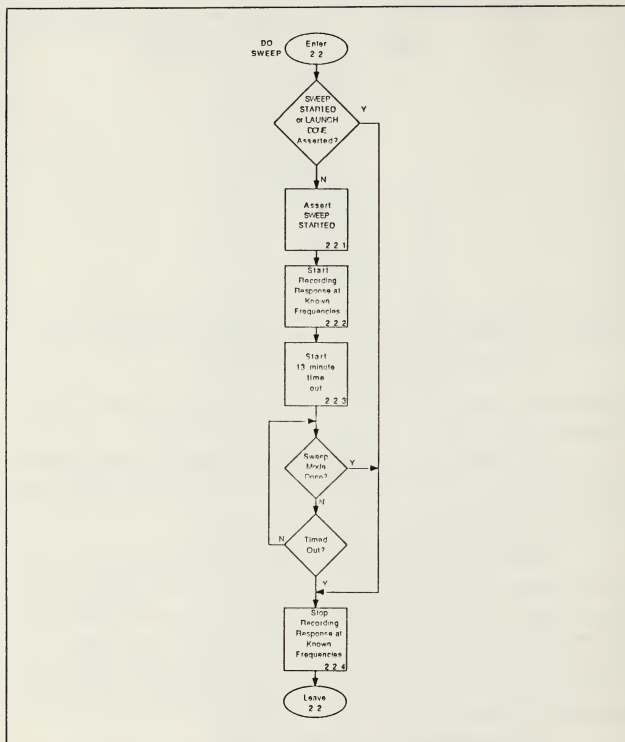


Figure 24. Flowchart 2.2: This flowchart shows the steps entailed in performing the *sweep* phase of the experiment.

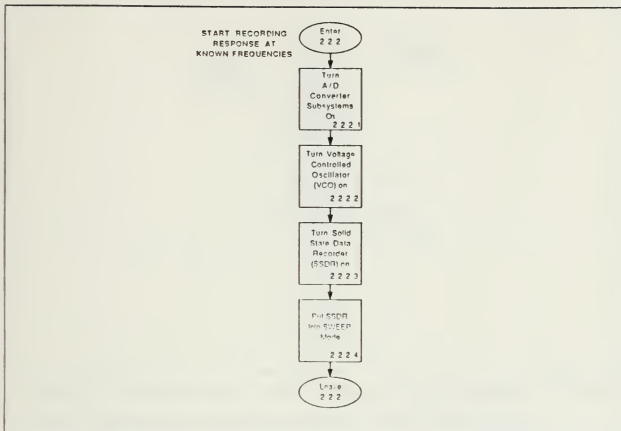


Figure 25. Flowchart 2.2.2: This flowchart shows the steps entailed in initiating the recording of known frequencies during the *sweep* phase of the experiment.

h. Wait for APUs to Start or for Launch Indications

Flowchart 2.3 in Figure 27 on page 55 is a more detailed look at block 2.3 of Flowchart 2 in Figure 22 on page 50. There are two possible indications of a launch. One is a signal from the Vibration-activated Launch Detector circuit. The other is a signal from the barometric switches. If either of these is present, the flag **LAUNCHED** is asserted. Otherwise, the controller will check to see if the Auxiliary Power Units (APUs) have been detected. If so, the flag **APUs ON** is asserted. If no indications are present, the controller will continue looking for them indefinitely.

i. Do Scroll

Flowchart 2.4 in Figure 28 on page 56 is a more detailed look at block 2.4 of Flowchart 2 in Figure 22 on page 50. It shows the steps entailed in performing the

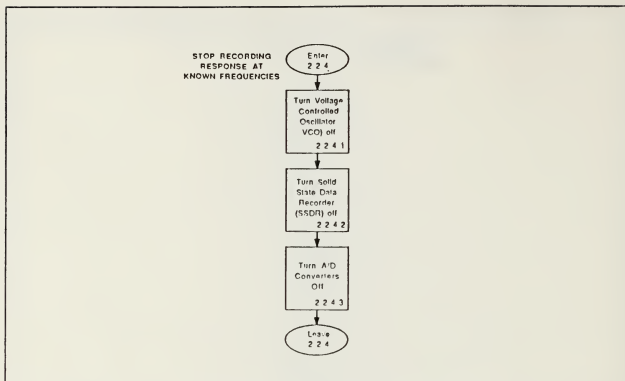


Figure 26. Flowchart 2.2.4: This flowchart shows the steps entailed in stopping the recording of known frequencies during the *sweep* phase of the experiment.

scroll phase. Power is applied to the Solid State Data Recorder (SSDR), and it is then commanded to enter *scroll* mode.

The mission will be scrubbed if no launch occurs within seven minutes after the Auxiliary Power Units are started. We initiate a ten minute timeout, which is conservative. If at the end of ten minutes no launch indications have been detected, the program aborts; otherwise, it asserts the **LAUNCHED** flag.

j. Abort

Flowchart 2.4.4 in Figure 29 on page 57 is a more detailed look at block 2.4.4 of Flowchart 2.4 in Figure 28 on page 56. It shows that when the mission is deemed to have been aborted, power is removed from all subsystems.

k. Do Launch

Flowchart 2.5 in Figure 30 on page 58 is a more detailed look at block 2.5 of Flowchart 2 in Figure 22 on page 50. The flowchart shows the steps entailed in performing the *launch* phase of the experiment. The first step is to determine whether

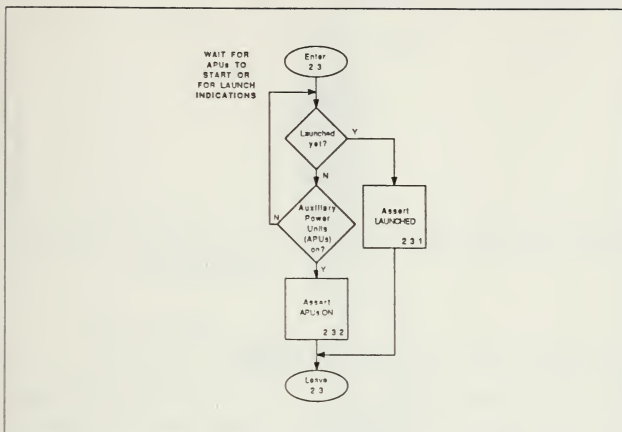


Figure 27. Flowchart 2.3: This flowchart shows the steps entailed in listening for the Auxiliary Power Units (APUs) while simultaneously waiting for indications of a launch.

this is necessary or not. If the launch was ever determined to have been completed in the past, the **LAUNCH DONE** will have been asserted then. In this case, it is not appropriate to perform this phase a second time and so all the blocks in this flowchart will be skipped. Otherwise, the Solid State Data Recorder (SSDR) is commanded to leave *scroll* mode and enter *launch* mode. Within two minutes of the time of launch, there will no longer be any air in the Space Shuttle's cargo bay. We initiate a three minute timeout so that, in the event that the SSDR fails to signal completion of the *launch* phase, the controller will not be stuck permanently in *launch* mode. The controller repeatedly checks either for a completion signal from the SSDR or for a timeout.

1. Check for a Completed Launch

Flowchart 2.5.3 in Figure 31 on page 59 is a more detailed look at block 2.5.3 of Flowchart 2.5 in Figure 30 on page 58. If the **LAUNCH DONE** flag ever was

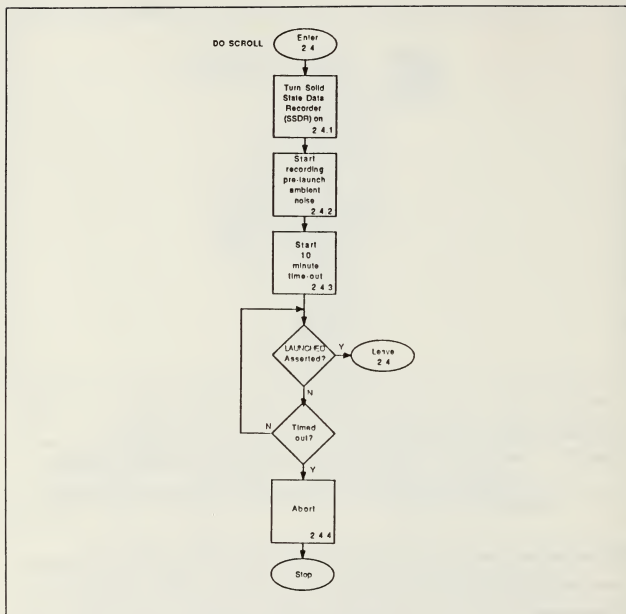


Figure 28. Flowchart 2.4: This flowchart shows the steps entailed in performing the *scroll* phase between the time the Auxiliary Power Units (APUs) start operating and the time that the launch is detected.

asserted, this block does nothing. However, if this flag was not previously asserted, the state of the barometric pressure switches is checked. If either of them has tripped, we have a positive indication of launch. It is then appropriate to set the LAUNCH DONE flag.

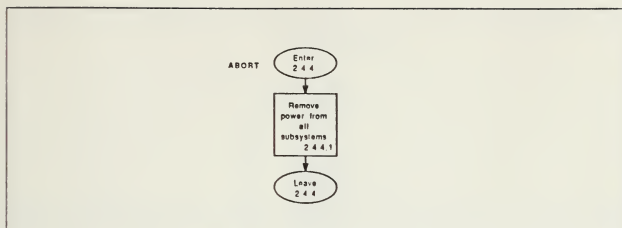


Figure 29. Flowchart 2.4.4: This flowchart shows what happens when an *abort* condition is detected.

m. Do Post-launch

Flowchart 2.6 in Figure 32 on page 60 is a more detailed look at block 2.6 of Flowchart 2 in Figure 22 on page 50. It shows the steps entailed in performing post-launch functions. These are the monitoring functions required after the Space Shuttle has left the earth's atmosphere. The first step is to remove power from all subsystems. A five minute timeout is then initiated. The effect of this is to permit system status to be recorded every five minutes. The heater subsystem is one of the subsystems which needs monitoring. A check is done to ensure that the launch has been recorded as complete. These two steps are repeated continually throughout each five minute period. At the completion of that period, current values of the temperature and voltage at various points are read and stored in the bubble memory log. A check also is made to see if the voltages on the buses are too low. If so, the post-launch processing ceases.

n. Monitor Heater Subsystem Operation

Flowchart 2.6.3 in Figure 33 on page 61 is a more detailed look at block 2.6.3 of Flowchart 2.6 in Figure 32 on page 60. It has two branches. In one, the temperature of the experimental apparatus is sufficiently high, in which case the heater subsystem is deactivated. In the other branch, the temperature is too low and the heater subsystem is activated.

o. Do Record

Flowchart 2.7 in Figure 34 on page 62 is a more detailed look at block 2.7 of Flowchart 2 in Figure 22 on page 50. It shows the steps entailed in putting the experiment into the *record* phase. These steps are, first, to put the Solid State Data Re-

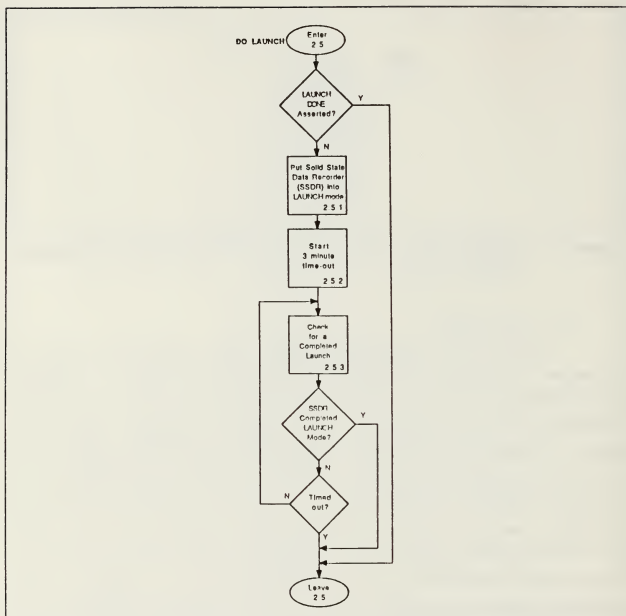


Figure 30. Flowchart 2.5: This flowchart shows the steps entailed in performing the *launch* phase of the experiment.

corder (SSDR) into the *launch* mode, and then to begin a 20 minute timeout, after which time the SSDR would have run out of memory in which to store recorded sound. The SSDR normally would inform the controller that it has completed operation before the timeout occurs. The timeout is meant to permit the controller to leave the *record* phase even if the SSDR fails to signal completion.

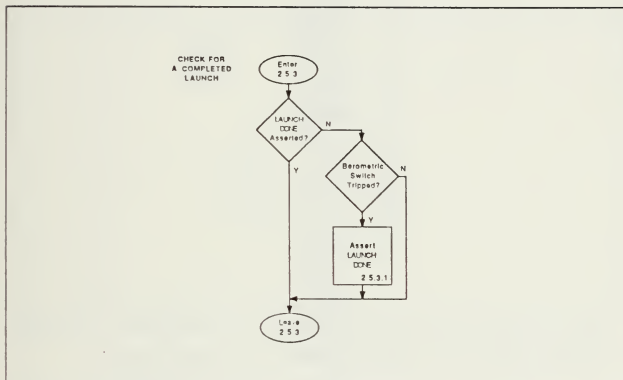


Figure 31. Flowchart 2.5.3: This flowchart shows the steps entailed in deciding whether or not a launch has occurred. The barometric switch is deemed to be the most reliable (and only convincing) indication of a launch.

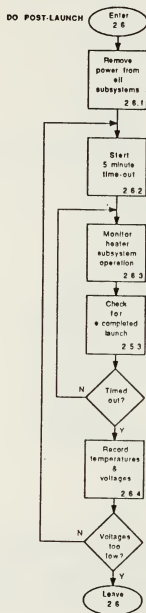


Figure 32. Flowchart 2.6: This flowchart shows the steps entailed in performing measurements and monitoring temperatures and voltages after the experiment is complete.

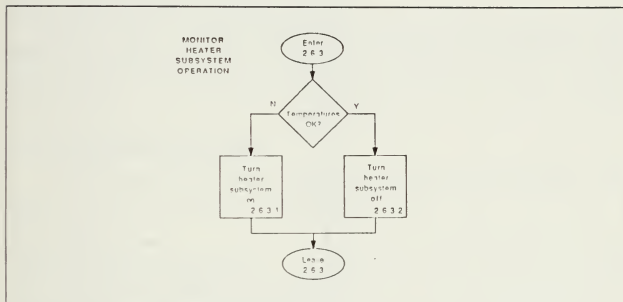


Figure 33. Flowchart 2.6.3: This flowchart shows the steps entailed in monitoring the temperatures of the bubble memory unit and maintaining that temperature above 10°C.

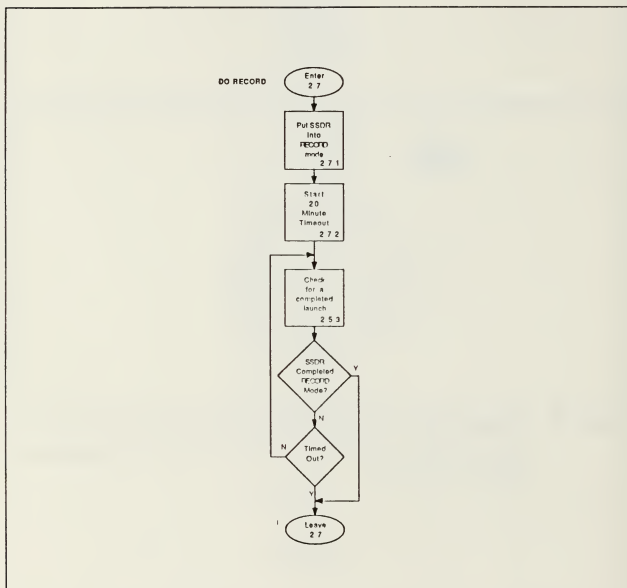


Figure 34. Flowchart 2.7: This flowchart shows the steps entailed in performing the *record* phase of the abridged experiment.

V. HOW TO GET THE EXPERIMENT READY FOR A LAUNCH

This chapter explains what must be done prior to the launch of the Space Shuttle to ensure that the experiment is performed correctly. The current status of the experiment is stored in page 0 of the bubble memory log. By setting appropriate information there, we can ensure that when power next is applied to the experimental apparatus, the correct sequence of steps is performed.

There are really two possible experiments to be performed: the unabridged experiment and the abridged one. The abridged experiment dispenses with the *sweep*, *scroll*, and *launch* phases of the experiment, replacing them with a single *record* phase. Which of these is to be run must be stored in page 0 before launch.

A. UNABRIDGED EXPERIMENT

Attach a terminal to the RS-232C interface and apply power to the experiment. The controller will present the following menu on the terminal.

- A Software reset.
- B Realtime clock functions.
- C Power relay switching functions.
- D Bubble memory test functions.
- E A/D converter functions.
- F Run experiment.
- G Perform port I/O.
- H Display contents of controller memory.
- I Examine or change the data logged in the bubble memory.
- Z Exit this menu.

Choose option (I). The following menu will be presented next.

- A Display page 0.
- B Display a page of the log.
- C Alter the contents of page 0.
- Z Exit this menu.

Choose option (C). The following menu will next be displayed.

- A Toggle 'sweepstarted' flag from TRUE to FALSE.
- B Toggle 'launchdone' flag from TRUE to FALSE.
- C Alter value of next available page from $0 \times 12 = 18$.
- D Alter value of next available half page from 1 to 0.
- E Toggle 'full_experiment' flag from TRUE to FALSE.
- F Specify the 'RECORD_start_time' (make this at least 12 hours before the present to permit RECORD mode to be initiated.)
- Z Exit this menu.

The menu may not look exactly like this, inasmuch as the flags and page numbers may vary. The objective, however, is to set the **sweepstarted** flag to FALSE; the **launchdone** flag to FALSE; the value of the next available page to 1; the value of the next available half-page to 0; the value of the **full_experiment** flag to TRUE. The value of the **RECORD_start_time** does not matter since the *record* phase is not performed in the unabridged experiment. If a value is already correct, it may be left alone. Only if it needs to be changed must the corresponding menu choice be made.

B. ABRIDGED EXPERIMENT

For the abridged experiment, follow the same steps as with the full experiment with the following differences. The **full_experiment** flag should be set to FALSE, and the **RECORD_start_time** should be set to some value at least 12 hours before launch. The value of the **sweepstarted** flag does not matter because the *sweep* phase is omitted in the abridged experiment.

C. BOTH VERSIONS OF THE EXPERIMENT

Once all the required choices have been made, the controller may be shut off and the terminal should be removed. The next time power is applied, the controller will discover that no terminal is attached, it will consult the values last stored in page 0 to see what it should do, and it will perform the experiment according to those values.

VI. TESTING OF THE SOFTWARE

Every module of the software has been tested individually for correctness. The integrated control program also has been tested exhaustively, but because the hardware has not yet been completely integrated, there is a limit to what could be accomplished. This chapter discusses how the testing has been performed and suggests further tests to be done once hardware integration is complete.

The following hardware components have been completed and have been successfully operated by the integrated software:

1. Bubble memory for the experiment's log.
2. Terminal.
3. Real Time Clock.
4. On-board Analog-to-digital (A/D) converter.
5. Voltage Controlled Oscillator.
6. Power Control Subsystem.

In addition, a preliminary design of the Solid State Data Recorder (SSDR) was tested by Kuebler [Ref. 9]. His tests included a demonstration that the off-board analog-to-digital converters associated with the SSDR functioned correctly, that the SSDR properly stored the acoustical data provided to it by the off-board analog-to-digital converters, that this data could be retrieved from the SSDR, and that an analysis could then be performed on the data. The final version of the SSDR has not yet been completed, and in the testing of the software described in this thesis, no attempt has been made to emulate its performance. However, the controller dutifully sends commands to it and makes repeated (unsuccessful) attempts to read its status information. It also notes its inability to get correct responses in the log.

The software has been tested many times under various conditions, both with and without a terminal attached. A test requires the initialization of flags in page 0 of the bubble memory log in advance. How to do this is explained in Chapter V. HOW TO GET THE EXPERIMENT READY FOR A LAUNCH on page 63. There are two ways to end a test, depending on whether or not a terminal is attached.

1. If there is a terminal attached, pressing CTRL Y will interrupt the experiment and present the highest level menu in the diagnostic subsystem. The experiment can be resumed by making choice

Z. Exit this menu.

To terminate the experiment completely, make choice

A. Software Reset.

2. If there is no terminal attached, simply remove power from the system. This is, in fact, how the experiment will be terminated at the end of a space flight (if the batteries last long enough.) Attaching a terminal and applying power will put the control program into the diagnostic subsystem.

The results of the experiment can be evaluated by making menu choice

I Examine or change the data logged in the bubble memory.

The following is a list of the various conditions under which the experiment has been tested. The conditions are listed as applying and not applying, where this is meaningful. All these conditions resulted in satisfactory performance by the controller.

1. Terminal present; terminal absent.
2. Unabridged experiment to be performed; abridged experiment to be performed.
3. *Sweep* phase previously performed; *sweep* phase not previously performed.
4. Launch had occurred previously; launch had not occurred previously.
5. Bubble memory space exhausted during test. The controller correctly ceased trying to store more data and continued to operate normally without logging its actions. This could not be verified without the terminal attached, for with neither a terminal nor a bubble memory log, the controller does not generate enough outputs for proper verification of its performance. There is no reason to suppose that the results would be different with the terminal removed, however.
6. Temperature out of limits. Operation of the bubble memory ceased. Power was applied to and removed from the heater subsystem by commands to the power control subsystem. These commands were issued by the subprogram responsible for monitoring the heater's operation, namely `monitor_heaters()`. Although the heater subsystem itself has not yet been completed, the associated power relays switched correctly.
7. Detection of the Auxiliary Power Units (APUs) by the Matched Filter was and was not emulated by toggle switch. The controller responded correctly in both cases.
8. Detection of launch by the Vibration-activated Launch Detector was and was not emulated by toggle switch. The controller responded correctly in both cases.
9. Activation of the Barometric Pressure Switches was and was not emulated by toggle switch. In the absence of activation of the barometric pressure switches, the presumption is that launch did not occur. The controller responded correctly in both cases.
10. Power was removed abruptly at various points in the procedure and then was restored. The controller recovered in the desired manner.

After each test, the contents of the bubble memory log were examined to see what steps in the experiment had been performed. Two obvious differences were found between the system's behavior with and without a terminal attached. One of these was that with a terminal attached, the diagnostic messages issued during the performance of the experiment were visible. The other was that with a terminal attached, the diagnostic subsystem did not get control. No other difference could be found between system behavior with and without a terminal attached.

There is every reason to believe that the control program will work as well in the Space Shuttle as it has in the lab. The same tests should be performed again once the hardware is completely integrated. The only tests which have not already been done are those associated with the operation of the Solid State Data Recorder (SSDR). The interface with this device is very simple. The controller sends commands and reads status information. The SSDR is otherwise completely independent of the controller. We have already ascertained that the application of power to the SSDR through the power control subsystem is done correctly. We have also verified correct response to a failure of the SSDR to perform as expected. The only thing we have not tested is the response of the control program to *correct* responses from the SSDR. Since the response amounts to making a note of the correct response in the bubble memory log, we do not anticipate any difficulty in this area. The ability of the controller to send commands to the SSDR has been tested, although the response of the SSDR to these commands cannot be evaluated until the final version of the SSDR is complete.

The means of extraction of data from the bubble memory log are not very elaborate at this point. Data can be extracted for two experiment steps at one time by providing the number of the page in the bubble memory log whose contents are required. No capability has been provided for rapid extraction of all data to, say, a microcomputer. This does not pose a serious problem, but data extraction would be facilitated by providing subroutines to do it quickly.

VII. CONCLUSIONS

The Vibro-acoustic Experiment was the first experiment produced at the Naval Postgraduate School for inclusion in a Space Shuttle mission. In view of the considerable technical hurdles it presented, it is fair to say it has been a very ambitious project. It has included many disciplines, such as mechanical engineering, thermal engineering, digital electronics, analog electronics, acoustics, bubble memory technology, autonomous computer control, software development, a matched filter for detecting an impending space shuttle launch, power control, and computer-aided design and manufacturing (CAD CAM).

This thesis has concerned itself with overall control of the experiment and with a description of one subsystem, the matched filter. It was not possible to do this without considering the experiment as a whole. From the author's personal standpoint, this has been very gratifying. We have used a high-level programming language to do the bulk of the programming of a microprocessor-based controller, thus avoiding the labor-intensive burden of assembly language coding in most areas of the program. We have effectively integrated this code with the little assembly code required. We have written drivers for assorted hardware devices, such as a terminal, a bubble memory module and a real-time clock, thus demonstrating the close association between hardware and the software which makes that hardware more than a glorified paper-weight. We have used structured programming techniques throughout, thus aiding the design process, as well as the understanding of the documentation represented in part by this thesis. We have created a conveniently-used software development system, making it straightforward to edit the source files, compile or assemble them, link the object modules, list source files on a printer, and place the executable file in EPROM.

This work may benefit others who would like to implement other applications. We have made it possible for them quickly to get a device controller programmed and operating, since so many standard controller functions already exist. The C programming language is highly portable, so much of this work would apply even with a new hardware design, using, say, a faster microprocessor with more memory.

Two other projects have already benefitted from the work done here. In one, the experimenters plan to obtain voltage *versus* current for solar cells in space. In less than two months, the complete control program for that new application was designed and

tested using the same controller and much of the software we have described in this thesis.

In another project now underway, the author is involved in the experimental evaluation of a thermo-acoustic refrigerator in space. The details of the experimental procedure in both these cases differ, but the overall fundamentals of control of an experiment are the same. Consequently, much of the software described in this thesis can be used without any modification.

For anyone who wants to build a controller with modest requirements for speed and RAM, the controller we have used in the Vibro-acoustic Experiment would be suitable. By using the work described in this thesis, one can avoid the unpleasant burden of starting from scratch. The bubble memory module provides a further 500K bytes of random access, non-volatile memory for whatever purpose might be required.

Work remaining to be done is:

1. Design, build and test an improved controller which would operate with more memory and greater speed.
2. Convert the existing software to run on the new controller. This would entail replacing the start-up code and other machine-dependent assembly code, and re-compiling the C language source code using a compiler which would generate machine-language code for the new target machine. Software Development Systems, Inc., makes C language cross-compilers which would allow most of the existing set-up to be preserved intact.
3. Modify the existing bubble memory drivers to permit storage of files. At the moment, the bubble memory is regarded as a linear list of 64-byte chunks of memory. Greater usefulness would result from the provision of a file subsystem.
4. Produce a manual for other potential users of the software and hardware that would make it easy to get new applications up and running quickly. This could be supplemented by improved routines to facilitate the generation of executable code. At the moment, these routines are specific to the Vibro-acoustic Experiment in that they always look in the subdirectory \vibro\contrlr for the files they need. In general, they should permit any subdirectory to be used to hold the files. The experiment to evaluate solar cell performance used a series of files and subdirectories whose structure exactly mimicked the set-up described in this thesis, but no routines have been written to set these files up automatically.
5. Develop or acquire a software maintenance system. Such a system would provide better management of successive versions of the control program. It would also include a data dictionary to provide a definition of all variables and functions, along with a complete cross-reference showing every place these objects were used. What would the data dictionary gain us? It often happens that in the course of making changes, we inadvertently affect other parts of a system. The data dictionary would make it possible to find all those places and take appropriate action.

Microprocessors have now been in use for just a little over 15 years. The use of compiled programs to operate them is an even more recent development, since compiled programs generally take up more memory space than assembled programs, and abundant memory at low prices is available. There are two distinct advantages to using a compiler:

1. The code is easier both to write and to understand. This makes it easier to get applications running, and to modify them later, even if team members change over a period of time.
2. The code is much faster to create. This often makes the difference between success and failure, since time can be critical, particularly in an educational environment.

Often these advantages outweigh the disadvantages:

1. The code tends to take more memory.
2. The code tends to execute more slowly.

In the case of the Vibro-acoustic Experiment, these factors were of no great consequence, except as already noted in conjunction with the bubble memory module. We were forced to use assembly code to operate the bubble memory in order to keep up with the data transfer rate demanded by it. Had there been an adequate buffer in that hardware, this extra effort could have been avoided.

In fact, this last point makes it clear that good hardware design can greatly reduce the effort (*i.e.*, the costs) associated with software development. It is hard to believe that the job of implementing a 64-byte buffer in hardware would be much more difficult than that of implementing a 40-byte buffer. Had Intel done this, we would have been spared months of development difficulties, during which we did not understand the reason why the C code did not work.

Finally, the use of a compiled language which is highly portable (in particular a compiled language such as C) can help protect the investment of time and money in software which is otherwise threatened as obsolete hardware is replaced by improved hardware. In the software industry, endless conversions from one hardware system to a new one seem to be a permanent nuisance. The ability to take old programs and make them work on new hardware simply by recompiling them is attractive.

Another potentially useful step would be to implement the controller software using Ada, the Department of Defense compiled language now mandated for embedded software in all purchased systems. This would have the advantage that Ada programmers

are likely to become more and more numerous over time. This would help keep the work already done both current and useful.

APPENDIX A. DERIVATION OF DESIGN EQUATIONS FOR THE MATCHED FILTER

This appendix presents derivations and proofs of results used elsewhere in this thesis.

A. BIQUADRATIC FILTERS USING TWO OPERATIONAL AMPLIFIERS

Figure 7 on page 27 shows the topology of a generalized biquadratic filter using two operational amplifiers. This topology is taken from Michael [Ref. 15]. In this figure, Y denotes an admittance (the reciprocal of an impedance). Michael gives ideal transfer functions in the s -domain from the node marked V_{in} to the nodes marked V_1 , V_2 , and V_3 . We will have occasion to use the first two of these. Since Michael does not provide derivations for these functions, they are provided below. Note that the term "ideal" implies the use of operational amplifiers of infinite gain. While no such operational amplifiers in fact exist, there do exist operational amplifiers of very large gain, and the approximation is practical in many circumstances, especially at the low frequencies used in the Vibro-acoustic Experiment (*i.e.*, the 600 Hz tone from the Auxiliary Power Unit). For example, the open-loop gain of the LF-444 operational amplifier is more than 60 dB at a frequency $f = 600$ Hz.

From the Kirkoff current law, and referring to Figure 7 on page 27, we have

$$I_2 = I_5 + I_6 \quad (35)$$

$$I_7 = I_8 - I_4 \quad (36)$$

$$I_1 = -I_3. \quad (37)$$

We can find the currents I_i through I_8 by applying Ohm's law.

$$I_1 = (V_1 - V_4)Y_1 \quad (38)$$

$$I_2 = (V_1 - V_5)Y_2 \quad (39)$$

$$I_3 = (V_2 - V_4)Y_3 \quad (40)$$

$$I_4 = (V_2 - V_3)Y_4 \quad (41)$$

$$I_5 = (V_5 - V_{IN})Y_5 \quad (42)$$

$$I_6 = V_5 Y_6 \quad (43)$$

$$I_7 = (V_{IN} - V_3) Y_7 \quad (44)$$

$$I_8 = V_3 Y_8. \quad (45)$$

By substituting these currents into equations (35) through (37), we obtain the following three, independent equations.

$$(V_1 - V_5) Y_2 = (V_5 - V_{IN}) Y_5 + V_5 Y_6 \quad (46)$$

$$(V_{IN} - V_3) Y_7 = V_3 Y_8 + (V_3 - V_2) Y_4 \quad (47)$$

$$(V_1 - V_4) Y_1 = (V_4 - V_2) Y_3. \quad (48)$$

Collecting these terms yields

$$V_5(Y_2 + Y_5 + Y_6) = V_1 Y_2 + V_{IN} Y_5 \quad (49)$$

$$V_3(Y_4 + Y_7 + Y_8) = V_2 Y_4 + V_{IN} Y_7 \quad (50)$$

$$V_4(Y_1 + Y_3) = V_1 Y_1 + V_2 Y_3. \quad (51)$$

In an ideal operational amplifier, the inputs have equal voltages, so we can write $V_3 = V_4 = V_5$. Rewriting the equations with V_3 in place of V_4 and V_5 gives

$$V_3(Y_2 + Y_5 + Y_6) = V_1 Y_2 + V_{IN} Y_5 \quad (52)$$

$$V_3(Y_4 + Y_7 + Y_8) = V_2 Y_4 + V_{IN} Y_7 \quad (53)$$

$$V_3(Y_1 + Y_3) = V_1 Y_1 + V_2 Y_3. \quad (54)$$

These equations can be readily solved by placing them in matrix form first.

$$\begin{bmatrix} Y_2 + Y_5 + Y_6 & 0 & -Y_2 \\ Y_4 + Y_7 + Y_8 & -Y_4 & 0 \\ Y_1 + Y_3 & -Y_3 & -Y_1 \end{bmatrix} \begin{bmatrix} V_3 \\ V_2 \\ V_1 \end{bmatrix} = \begin{bmatrix} Y_5 \\ Y_7 \\ 0 \end{bmatrix} V_{IN}. \quad (55)$$

Now interchange the positions of V_2 and V_3 .

$$\begin{bmatrix} 0 & Y_2 + Y_5 + Y_6 & -Y_2 \\ -Y_4 & Y_4 + Y_7 + Y_8 & 0 \\ -Y_3 & Y_1 + Y_3 & -Y_1 \end{bmatrix} \begin{bmatrix} V_2 \\ V_3 \\ V_1 \end{bmatrix} = \begin{bmatrix} Y_5 \\ Y_7 \\ 0 \end{bmatrix} V_{IN}. \quad (56)$$

Next we move row 3 to the top.

$$\begin{bmatrix} -Y_3 & Y_1 + Y_3 & -Y_1 \\ 0 & Y_2 + Y_5 + Y_6 & -Y_2 \\ -Y_4 & Y_4 + Y_7 + Y_8 & 0 \end{bmatrix} \begin{bmatrix} V_2 \\ V_3 \\ V_1 \end{bmatrix} = \begin{bmatrix} 0 \\ Y_5 \\ Y_7 \end{bmatrix} V_{IN}. \quad (57)$$

Next, multiply row 1 by Y_4 , row 3 by Y_3 , and subtract the latter product from the former to generate a new row 3.

$$\begin{bmatrix} -Y_3 & Y_1 + Y_3 & -Y_1 \\ 0 & Y_2 + Y_5 + Y_6 & -Y_2 \\ 0 & -Y_3(Y_4 + Y_7 + Y_8) + Y_4(Y_1 + Y_3) & -Y_1 Y_4 \end{bmatrix} \begin{bmatrix} V_2 \\ V_3 \\ V_1 \end{bmatrix} = \begin{bmatrix} 0 \\ Y_5 \\ -Y_3 Y_7 \end{bmatrix} V_{IN}. \quad (58)$$

Now multiply row 2 by $[-Y_3(Y_4 + Y_7 + Y_8) + Y_4(Y_1 + Y_3)]$, row 3 by $(Y_2 + Y_5 + Y_6)$, and subtract the latter product from the former to generate yet another row 3.

$$\begin{aligned} & \begin{bmatrix} -Y_3 & Y_1 + Y_3 & -Y_1 \\ 0 & Y_2 + Y_5 + Y_6 & -Y_2 \\ 0 & 0 & -Y_1 Y_4(Y_2 + Y_5 + Y_6) + Y_2[-Y_3(Y_4 + Y_7 + Y_8) + Y_4(Y_1 + Y_3)] \end{bmatrix} \begin{bmatrix} V_2 \\ V_3 \\ V_1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ Y_3 \\ -Y_3 Y_7(Y_2 + Y_5 + Y_6) - Y_3[-Y_3(Y_4 + Y_7 + Y_8) + Y_4(Y_1 + Y_3)] \end{bmatrix} V_{IN}. \end{aligned} \quad (59)$$

From row 3, we can see that

$$\frac{V_1}{V_{IN}} = \frac{-Y_3 Y_7(Y_2 + Y_5 + Y_6) - Y_3[-Y_3(Y_4 + Y_7 + Y_8) + Y_4(Y_1 + Y_3)]}{-Y_1 Y_4(Y_2 + Y_5 + Y_6) + Y_2[-Y_3(Y_4 + Y_7 + Y_8) + Y_4(Y_1 + Y_3)]} \quad (60)$$

$$\frac{V_1}{V_{IN}} = \frac{-Y_3 Y_7(Y_2 + Y_6) - Y_3 Y_3 Y_7 + Y_3 Y_4 Y_3 + Y_3 Y_3 Y_7 + Y_3 Y_3 Y_8 - Y_1 Y_4 Y_3 - Y_3 Y_4 Y_5}{-Y_1 Y_4(Y_5 + Y_6) - Y_1 Y_2 Y_4 - Y_2 Y_3 Y_4 - Y_2 Y_3 Y_7 - Y_2 Y_3 Y_8 + Y_1 Y_2 Y_4 + Y_2 Y_3 Y_4}. \quad (61)$$

Many terms in the numerator and in the denominator of this expression add to zero, so the transfer function is

$$\frac{V_1}{V_{IN}} = \frac{-Y_3 Y_7(Y_2 + Y_6) + Y_3 Y_3 Y_8 - Y_1 Y_4 Y_5}{-Y_1 Y_4(Y_5 + Y_6) - Y_2 Y_3 Y_7 - Y_2 Y_3 Y_8} \quad (62)$$

$$\frac{V_1(s)}{V_{IN}(s)} = \frac{Y_1 Y_4 Y_5 + Y_3 Y_7 (Y_2 + Y_6) - Y_3 Y_5 Y_8}{Y_1 Y_4 (Y_5 + Y_6) + Y_2 Y_3 (Y_7 + Y_8)}. \quad (63)$$

This completes the derivation of $\frac{V_1(s)}{V_{IN}(s)}$. To derive the transfer function $\frac{V_2(s)}{V_{IN}(s)}$, we modify equation (55) by placing the variable V_2 in the last position.

$$\begin{bmatrix} -Y_2 & Y_2 + Y_5 + Y_6 & 0 \\ 0 & Y_4 + Y_7 + Y_8 & -Y_4 \\ -Y_1 & Y_1 + Y_3 & -Y_3 \end{bmatrix} \begin{bmatrix} V_1 \\ V_3 \\ V_2 \end{bmatrix} = \begin{bmatrix} Y_5 \\ Y_7 \\ 0 \end{bmatrix} V_{IN}. \quad (64)$$

Proceeding as before, we multiply row 3 by $-Y_2$, row 1 by $-Y_1$, and subtract the former product from the latter to generate a new row 3.

$$\begin{bmatrix} -Y_2 & Y_2 + Y_5 + Y_6 & 0 \\ 0 & Y_4 + Y_7 + Y_8 & -Y_4 \\ 0 & Y_1(Y_2 + Y_5 + Y_6) - Y_2(Y_1 + Y_3) & Y_2 Y_3 \end{bmatrix} \begin{bmatrix} V_1 \\ V_3 \\ V_2 \end{bmatrix} = \begin{bmatrix} Y_5 \\ Y_7 \\ Y_1 Y_5 \end{bmatrix} V_{IN}. \quad (65)$$

Next multiply row 2 by $[Y_1(Y_2 + Y_5 + Y_6) - Y_2(Y_1 + Y_3)]$, row 3 by $(Y_4 + Y_7 + Y_8)$, and subtract the latter product from the former to yield a new row 3.

$$\begin{bmatrix} -Y_2 & Y_2 + Y_5 + Y_6 & 0 \\ 0 & Y_4 + Y_7 + Y_8 & -Y_4 \\ 0 & 0 & Y_2 Y_3 (Y_4 + Y_7 + Y_8) + Y_4 [Y_1(Y_2 + Y_5 + Y_6) - Y_2(Y_1 + Y_3)] \end{bmatrix} \begin{bmatrix} V_1 \\ V_3 \\ V_2 \end{bmatrix} = \begin{bmatrix} Y_5 \\ Y_7 \\ Y_1 Y_5 (Y_4 + Y_7 + Y_8) - Y_2 [Y_1(Y_2 + Y_5 + Y_6) - Y_2(Y_1 + Y_3)] \end{bmatrix} V_{IN}. \quad (66)$$

From row 3 we can see that

$$\frac{V_2}{V_{IN}} = \frac{Y_1 Y_5 (Y_4 + Y_7 + Y_8) - Y_1 Y_7 (Y_2 + Y_5 + Y_6) + Y_2 Y_7 (Y_1 + Y_3)}{Y_2 Y_3 (Y_4 + Y_7 + Y_8) + Y_1 Y_4 (Y_2 + Y_5 + Y_6) - Y_2 Y_4 (Y_1 + Y_3)}. \quad (67)$$

As before, many terms in both the numerator and the denominator add to zero, so

$$\frac{V_2}{V_{IN}} = \frac{Y_1 Y_5 (Y_4 + Y_8) + Y_2 Y_3 Y_7 - Y_1 Y_6 Y_7}{Y_2 Y_3 (Y_7 + Y_8) + Y_1 Y_4 (Y_5 + Y_6)}. \quad (68)$$

It is possible, using the same method illustrated in both these cases, to develop a transfer function $\frac{V_3(s)}{V_{IN}(s)}$. We have no use for this particular function in the present context, however, so the derivation is omitted.

B. HIGH-PASS NOTCH FILTER

The equation of a notch filter is given in equation (6), repeated here.

$$F(s) = \frac{s^2 + \omega_z^2}{s^2 + \left(\frac{\omega_p}{Q_p}\right)s + \omega_p^2}. \quad (6)$$

Michael [Ref. 15] shows how to use the generalized configuration of Figure 7 on page 27 to implement this function in the case where $\omega_z = \omega_p$, which is the case for a symmetric notch filter. However, he does not show how to implement an asymmetrical notch filter, in which $\omega_p \neq \omega_z$.

We can do so by using equation (63) and making the following choices for the admittances Y_1 through Y_8 .

$$Y_1 = C_a \quad (69)$$

$$Y_2 = sC_a \quad (70)$$

$$Y_3 = C_b \quad (71)$$

$$Y_4 = Y_5 = G = \frac{1}{R} \quad (72)$$

$$Y_6 = 0 \quad (73)$$

$$Y_7 = sC_b \quad (74)$$

$$Y_8 = \frac{G}{Q_p} \quad (75)$$

where we pick

$$\omega_p = \frac{G}{C_b} \quad (76)$$

and

$$\frac{C_b}{C_a} = Q_p \left[1 - \left(\frac{\omega_z}{\omega_p} \right)^2 \right]. \quad (77)$$

Proof

$$\begin{aligned} \frac{V_1}{V_{IN}} &= \frac{C_a G^2 + s^2 C_a C_b^2 - \frac{C_b G^2}{Q_p}}{C_a G^2 + s^2 C_a C_b^2 + s C_a C_b \frac{G}{Q_p}} \\ &= \frac{s^2 + \left(\frac{G}{C_b} \right)^2 - \frac{G^2}{C_a C_b Q_p}}{s^2 + \left(\frac{G}{C_b Q_p} \right) s + \left(\frac{G}{C_b} \right)^2} \\ &= \frac{s^2 + \left(\frac{G}{C_b} \right)^2 \left[1 - \frac{C_b}{C_a Q_p} \right]}{s^2 + \left(\frac{G}{C_b Q_p} \right) s + \left(\frac{G}{C_b} \right)^2} \\ &= \frac{s^2 + \omega_p^2 \left[1 - \left(1 - \left[\frac{\omega_z}{\omega_p} \right]^2 \right) \right]}{s^2 + \left(\frac{\omega_p}{Q_p} \right) s + \omega_p^2} \\ &= \frac{s^2 + \omega_z^2}{s^2 + \left(\frac{\omega_p}{Q_p} \right) s + \omega_p^2}. \end{aligned} \quad (78)$$

Using resistance values instead of conductance values in equations (76) and (77), we get

$$\frac{C_b}{C_a} = Q_p \left[1 - \left(\frac{\omega_z}{\omega_p} \right)^2 \right]$$

(79)

and

$$R = \frac{1}{\omega_p C_b} . \quad (80)$$

C. LOW-PASS NOTCH FILTER

To get a low-pass notch filter, we use equation (68) and pick admittances as follows:

$$Y_1 = Y_5 = G_b \quad (81)$$

$$Y_2 = Y_7 = sC \quad (82)$$

$$Y_3 = Y_4 = G_a \quad (83)$$

$$Y_6 = 0 \quad (84)$$

$$Y_8 = \frac{G_b}{Q_p} \quad (85)$$

where we pick

$$\omega_p = \frac{G_b}{C} \quad (86)$$

and

$$\frac{G_b}{G_a} = Q_p \left[\left(\frac{\omega_z}{\omega_p} \right)^2 - 1 \right] . \quad (87)$$

$$\begin{aligned}
 \frac{V_2}{V_{IN}} &= \frac{G_b^2 \left(G_a + \frac{G_b}{Q_p} \right) + s^2 C^2 G_a}{G_b^2 G_a + s^2 C^2 G_a + sC \frac{G_b G_a}{Q_p}} \\
 &= \frac{s^2 + \left(\frac{G_b}{C} \right)^2 + \left(\frac{G_b}{C} \right)^2 \frac{G_b}{G_a Q_p}}{s^2 + \left(\frac{G_b}{C Q_p} \right) s + \left(\frac{G_b}{C} \right)^2} \\
 &= \frac{s^2 + \left(\frac{G_b}{C} \right)^2 \left[1 + \frac{G_b}{G_a Q_p} \right]}{s^2 + \left(\frac{G_b}{C Q_p} \right) s + \left(\frac{G_b}{C} \right)^2} \quad (88) \\
 &= \frac{s^2 + \omega_z^2 \left[1 + \left(\frac{\omega_z}{\omega_p} \right)^2 - 1 \right]}{s^2 + \left(\frac{\omega_p}{Q_p} \right) s + \omega_p^2} \\
 &= \frac{s^2 + \omega_z^2}{s^2 + \left(\frac{\omega_p}{Q_p} \right) s + \omega_p^2} .
 \end{aligned}$$

Converting equations (86) and (87) to use resistances instead of conductances, we get the design equations

$$\boxed{\frac{R_a}{R_b} = Q_p \left[\left(\frac{\omega_z}{\omega_p} \right)^2 - 1 \right]} \quad (89)$$

and

$$\boxed{C = \frac{1}{R_b \omega_p}} . \quad (90)$$

D. A SECOND-ORDER, LOW-PASS FILTER USING ONLY ONE OPERATIONAL AMPLIFIER

Figure 14 on page 36 is the schematic of a generalized second-order, low-pass filter. The general equation of a low-pass biquadratic filter is given by [Ref. 12 : p. 16] as

$$\frac{V_O(s)}{V_{IN}(s)} = \frac{\omega_p^2}{s^2 + \left(\frac{\omega_p}{Q_p}\right)s + \omega_p^2}. \quad (91)$$

This transfer function can be realized by the schematic in Figure 14 on page 36 if we make the following choices for the components.

$$\omega_p = \frac{1}{\sqrt{C_1 C_2 R_1 R_2}} \quad (92)$$

and

$$Q_p = \sqrt{\frac{C_1 R_1 R_2}{C_2}} \frac{1}{R_1 + R_2}. \quad (93)$$

Proof

$$I_1 = V_O s C_2 = \frac{V_a - V_O}{R_2} \quad (94)$$

$$I_2 = (V_a - V_O) s C_1 \quad (95)$$

$$I_3 = \frac{V_{IN} - V_a}{R_1} = I_2 + I_1. \quad (96)$$

From equation (94),

$$V_a = V_O [1 + s C_2 R_2] \quad (97)$$

or

$$V_a - V_O = V_O s C_2 R_2. \quad (98)$$

Thus

$$\begin{aligned}
I_3 &= (V_a - V_O)sC_1 + \frac{V_a - V_O}{R_2} \\
&= V_O[s^2C_1C_2R_2 + sC_2] \\
&= \frac{V_{IN} - V_a}{R_1} \\
&= \frac{V_{IN} - V_O[1 + sC_2R_2]}{R_1}.
\end{aligned} \tag{99}$$

We can manipulate this equation to obtain the transfer function.

$$V_O[s^2C_1C_2R_1R_2 + sC_2R_1 + 1 + sC_2R_2] = V_{IN} \tag{100}$$

$$\begin{aligned}
\frac{V_O}{V_{IN}} &= \frac{1}{s^2C_1C_2R_1R_2 + sC_2(R_1 + R_2) + 1} \\
&= \frac{\left(\frac{1}{C_1C_2R_1R_2}\right)}{\left[s^2 + \frac{(R_1 + R_2)}{C_1R_1R_2}s + \frac{1}{C_1C_2R_1R_2}\right]} \\
&= \frac{\omega_p^2}{\left[s^2 + \frac{\left(\frac{1}{C_1C_2R_1R_2}\right)}{\left[\sqrt{\frac{C_1R_1R_2}{C_2}}\left(\frac{1}{R_1 + R_2}\right)s + \omega_p^2\right]}\right]} \\
&= \frac{\omega_p^2}{s^2 + \left(\frac{\omega_p}{Q_p}\right) + \omega_p^2}.
\end{aligned} \tag{101}$$

If we choose $R_1 = R_2 = R$, then

$$\omega_p = \frac{1}{R\sqrt{C_1C_2}} \tag{102}$$

and

$$Q_p = \frac{1}{2} \sqrt{\frac{C_1}{C_2}}. \tag{103}$$

To design a filter to provide desired values of ω_p and Q_p , use the design equations which can easily be derived from the above equations.

1. Pick C_2 arbitrarily.
2. Let $C_1 = 4C_2Q_p^2$.
3. Let $R_1 = R_2 = R = \frac{1}{\omega_p \sqrt{C_1 C_2}}$.

APPENDIX B. CHOICE OF A SOFTWARE DEVELOPMENT SYSTEM

A. Z-80 ASSEMBLY LANGUAGE

The Vibro-acoustic Experiment has a fairly long history. Long before the author became associated with the project, two distinct choices for a software environment¹⁵ had already been made. Early in the project we used Z-80 assembly language for programming the controller. An ALTOS eight-bit microcomputer running under Digital Research Corporation's Control Program for Microcomputers (CP/M) was available. It included a Z-80 assembler (M80), a librarian (LIB80) and a linker (L80). However, the turnover in student personnel is rather high at any educational institution; the Naval Postgraduate School is no exception. Assembly language is often not the best choice for a project whose participants do not remain for the life of the project, since assembly language is not widely known, is not easy to learn, and is highly dependent on the architecture of the machine in which the final program is to operate. Many different architectures exist, and most machines have unique architectures. Even those who know how to program with assembly language often are averse to expending the vast amounts of time required to use it for any but the most trivial programs.

B. CP/M AND TOOLWORKS C

As a consequence of these facts, one of the early participants in the project successfully promoted a switch away from Z-80 assembly language to the C programming language. This high-level language includes powerful operators which make it easy to manipulate the bits within the bytes of the computer's memory, and so it can do almost anything that can be done in assembly language. The same ALTOS CP/M system happened to have Toolwork's C compiler on it, and so we used it.

When the author joined the project, little progress had been made in actually writing the control program. With Captain Frank Mazur, USMC, and Captain Ron Byrnes, USA, the author wrote much of the control program on the ALTOS under CP/M using the Toolwork's C compiler.

¹⁵ The term *software environment* refers to the computer, the operating system, the programming language, and all related software and hardware tools used to program a computer. The computer on which the development of software is done need not necessarily be the same one as that in which the completed program will reside and be executed. In the case of the Vibro-acoustic Experiment, it is not the same computer.

Doing so proved to be a frustrating business. The ALTOS was equipped with two eight-inch, single-sided, single-density, floppy diskette drives. These could contain only around 250 Kbytes of data. One drive had to contain a copy of the CP/M operating system at nearly all times. The ALTOS was quite slow by present standards; it was not uncommon for a compilation to take five minutes. Due to the limited disk space available, the output of the compiler (an 8080 assembly language source file¹⁶) had to be transferred to another diskette before assembly could proceed. Assembly typically consumed a further five minutes. The library program was quite inconvenient to use, but once the executable modules had been loaded successfully into a library, linking was straightforward. This was a comparatively quick two-minute process.

Our EPROM writing program was on an IBM-PC using Microsoft's Disk Operating System (MS DOS). Furthermore, that machine had only 5 1/4 inch diskettes. So we took our eight-inch floppies to a Zenith Z-100 that had both sizes of drives, where we converted the CP/M file containing executable code into an MS DOS file on a 5 1/4 inch diskette.

Finally, we loaded the executable program from the diskette into the EPROM-writing program and created the firmware.

C. MS/DOS AND UNIWARE C

It should not have taken us too long to tire of this agonizing procedure. In fact, it was over a year before we began seriously to search for an improvement in the form of a cross-compiler. We wanted a C-language compiler which would operate on an IBM PC using MS DOS, and which would generate Z-80 object code. Several were available. We selected the UniWare C Compiler package from Software Development Systems, 3110 Woodcreek Drive, Downers Grove, IL 60515. This product is a complete software development system. It includes a C compiler which produces Z-80 assembly code, a Z-80 assembler, a library manager to store object modules in a single MS DOS library file, a linker to convert a collection of object modules into an executable file, and a large collection of utility programs, useful for listing files, converting files from one format to another, and so on. The compiler implements the complete C language defined by

¹⁶ The Toolwork's C compiler generates 8080 assembly language source code. The NSC800 on the controller board can execute Z-80 code, a subset of the NSC800 instruction set, and the 8080 instruction set, which is itself a subset of the Z-80 instruction set. We had an assembler for Z-80 and 8080 code. We needed two Z-80 instructions not available in the 8080 instruction set. So we embedded Z-80 machine code in the 8080 assembly source created by the C compiler and executed the resultant module with an NSC800. It really was at least as complicated as it sounds!

Kernighan and Ritchie [Ref. 16]. It also includes enhancements similar, but not identical, to those proposed by the American National Standards Institute (ANSI).

It took a little time to convert from the old to the new system, but the results were well worth the effort. Because the performance of the IBM System 2 Model 80 on which we run this system is so much greater than that of the Altos, we are able to generate a new version of the controller program in much less time. The use of MS DOS also has provided significant benefits. We have made extensive use of hierarchical file directories in order to group files in a logical manner. We also use MS DOS batch files to minimize the amount of memory work necessary to execute such programs as the compiler and the linker.

The documentation supplied with the UniWare system [Ref. 17] is excellent. Unlike most C compilers, this one is not meant to produce executable code running under an operating system. For this reason, much of the standard library supplied with other C compilers is not applicable, and is not supplied. In particular, no library functions are provided to perform disk input or output. However, common output formatting routines such as `printf()` are included.

D. GENERATION OF FIRMWARE IN EPROM

We use the Intel program `pcpp` to load the completely linked program into EPROM. The UniWare software can create a symbol table showing what should go where. Armed with this list, one can load, install, and test the new version of the program in short order.

Details on the operation of this program are presented in Section 2. Getting the Executable Program into EPROM on page 146.

APPENDIX C. HOW THE UNIWARE SOFTWARE USES THE COMPUTER MEMORY

The UniWare software regards memory as comprised of a number of named regions. The C compiler itself creates five of these [Ref. 17: Compiler section, p.3]. These are the regions **code**, **string**, **const**, **data**, and **ram**. There are three further software regions: **reset**, **mbkrkam**, and **stack**. The purpose of each region is described below. The linker treats each region as a unit and places its contents in memory in contiguous storage locations. It decides how to do this based on instructions in the file `\vibro\contrlr\object\spec`. The order in which these regions appear in memory is specified in this file, and reflected both in Figure 19 on page 44, and in the order in which they are described here.

reset The Z-80 architecture specifies that the program code stored at memory location 0x0000 be executed whenever the microprocessor receives power or a hardware reset occurs. The **reset** region contains an appropriate start-up program. This program does the following:

1. It initializes the stack pointer to 0x0000. Whenever an item is stored in the stack, the stack pointer is first decremented. Thus, the stack pointer will initially be decremented to 0xffff, the first location in the stack, and will continue to grow downward in memory from this point.
2. It initializes the interrupt tables in such a manner that, should a spurious interrupt occur, the control program will restart from the beginning. It would be preferable to resume execution by simply returning from the interrupt. This would raise the unacceptable possibility, however, of an indefinite suspension of the execution of the program if some unpredictable cause led to the problem. While restarting has the disadvantage of totally disrupting matters, its compelling advantage is that execution resumes from a known state, barring a complete catastrophe.

code This region contains all program instructions generated by C and assembly language source code. It includes code to do the following things:

1. Program variables must be in RAM to be altered. In the C programming language it is possible to assign initial values to these variables at the time a program is compiled. These values must be placed in EPROM, since otherwise they would be lost. One of the routines in the **code** region is invoked at start-up time to copy initialized variables from their permanent locations in region **data** in EPROM into temporary locations in RAM. Thus in Figure 19 on page 44 region **data** appears in two locations, both in EPROM and in RAM.

2. The definition of the C programming language specifies that static¹⁷ and external¹⁸ variables which have no initial value specified in their declarations must be initialized by the compiler to the value 0 [Ref 16: p.198]. One of the routines in the **code** region is invoked at start-up time to put zeros in all RAM locations in region **ram**.
3. Another routine which is invoked at start-up time calls the C program **main()**. This is the highest level program in C. It calls subordinate routines to operate the controller and run the experiment itself.

string	Whenever the compiler finds a quoted character string in the source code, it places it in the string region. Since strings are treated as constants, they can be kept in EPROM. ¹⁹
const	Variables declared as const are regarded by the compiler as invariant, or constant, so it is reasonable to place them in EPROM.
data	This region contains variables whose initial values were specified at the time of compilation. These values are placed in EPROM by the linker so that they will not be lost when power is removed from the controller. However, variables must be in RAM when the program executes. During the start-up procedure, they are copied into RAM.
ram	This region contains variables whose initial values were not specified at the time of compilation. These are initialized to 0 at the time the program is first invoked, as specified in Kernighan and Ritchie [Ref. 16 : p. 198].
mbrkram	The UniWare C compiler provides a function mbrk() to permit a program to request storage at run time (<i>i.e.</i> , dynamically). The mbrkram region provides mbrk() with the storage it needs.
stack	The program stack is located here, at the top of memory.

The linker ensures that items within a region are stored contiguously. The compiler decides where to put these partitions in memory by examining a memory map provided in the specification file `\vibro\contrlr\object\spec`, listed in Section A. Filename `spec` on page 150. The format of this file is described in [Ref. 17: Link Editor Section, p. 7].

The memory map specifies that **reset** be loaded at address 0x0000, that the stack grow down in memory from address 0xffff, that EPROM is available from addresses 0x0000 through 0x5fff, and that RAM is available starting from address 0xe000 through 0xffff.

¹⁷ Static variables retain their values even after the program which declared them finishes executing.

¹⁸ External variables are declared in some module other than the one in which a program using these variables is defined.

¹⁹ In general, to modify strings a programmer must first place a copy of them into a variable. Dynamic variables are always located in RAM, since their contents are changeable.

APPENDIX D. HIERARCHICAL ORGANIZATION OF SOFTWARE FILES

All the software to control the Vibro-Acoustic Experiment is located in the file hierarchy illustrated in Figure 35 on page 89. Following is a description of the contents of each of these subdirectories.

A. SUBDIRECTORY \VIBRO\CONTRLR\HEADERS

This subdirectory contains header files for the C language source code. The header files allow numeric constants which are used in creating the program to be specified symbolically. By avoiding the use of “magic” numbers in the source code, the code is rendered much more readily understood. The header files also contain external declarations of the functions and variables contained within a module. Whenever one module needs to use the functions or variables of a different module, it can obtain correct declarations of them by including the appropriate header file using the C programming language `#include` directive.

B. SUBDIRECTORY \VIBRO\CONTRLR\CSOURCE

This subdirectory contains C language source code for the parts of the controller program written in the C programming language.²⁰

C. SUBDIRECTORY \VIBRO\CONTRLR\ASMSOURC

This subdirectory contains Z-80 assembly language source code. A few of the lowest level routines in the controller software have been written in assembly language, but only when there was no apparent way to write them in C (*e.g.*, `input()`, `output()`), or when the C compiler couldn't generate code which would execute rapidly enough (*e.g.*, `bubread()` and `bubwrite()`).

D. SUBDIRECTORY \VIBRO\CONTRLR\BATCH

This subdirectory contains a number of MS/DOS “batch” files. These contain sequences of commands which make it easier to compile programs, print listings of the source code, link object modules, and load executable modules into EPROMs.

²⁰ This comprises most of the controller software.

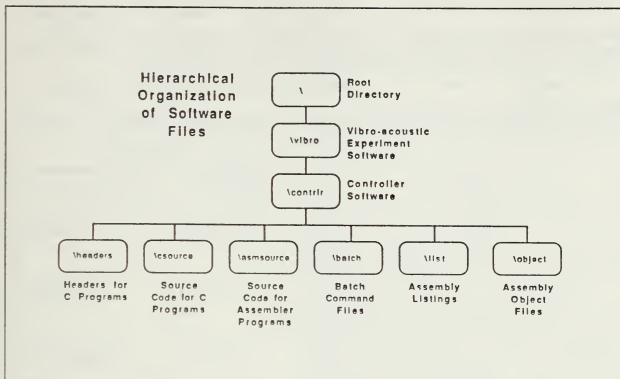


Figure 35. Hierarchical Organization of Software Files: The software files are grouped into several different sub-directories to facilitate finding and managing them.

E. SUBDIRECTORY \VIBRO\CONTRLR\LIST

This subdirectory contains output listings produced either by the C compiler or by the Z-80 assembler.²¹ Those created from C source code include that code in the form of comments to the Z-80 assembler. They are stored in this subdirectory only as a matter of convenience, in order that they not clutter up the directory listing of the sub-directories containing the source code.

F. SUBDIRECTORY \VIBRO\CONTRLR\OBJECT

This subdirectory contains object modules produced either by the C compiler or by the Z-80 assembler. They are stored in this subdirectory only as a matter of convenience, in order that they not clutter up the subdirectories containing the source code.

²¹ Those produced by the C compiler are actually assembly language listings produced by the Z-80 assembler. The latter is called by the C compiler.

This subdirectory also contains the link specification file `spec`. This file provides the linker with information needed to decide where the various regions of the program must be loaded. A number of global variables are set by this file at link time.

For details on how to use a link specification file, see the discussion in [Ref. 17: Link Editor Section, p. 7].

APPENDIX E. SUBROUTINES, IN ALPHABETICAL ORDER BY NAME

Table 8. SUBROUTINE INDEX: This table shows the names of the MS-DOS files in which each subroutine can be found. Subroutines are listed alphabetically by name.

SUBROUTINE	SOURCE FILE	PURPOSE
<code>ad_read()</code>	<code>expmnt.c</code>	Gets a character of data from the Analog-to-digital (A/D) Converter.
<code>adtoint()</code>	<code>expmnt.c</code>	Converts a character of raw data from the Analog-to-digital (A/D) Converter into an integer format with the more meaningful units of volts or degrees kelvin.
<code>alter_page0()</code>	<code>expmnt.c</code>	Permits the user to alter the contents of page 0 of the bubble memory. This is required in initializing the experiment.
<code>allow_ctrl_interrupts()</code>	<code>inout.c</code>	Processes the special characters CTRL S and CTRL Y when read from the keyboard. CTRL S is a toggle switch. The first time it is pressed, the display halts. The second time it is pressed, the display resumes. Subsequently its function alternates between these two. CTRL Y invokes the diagnostic subsystem.
<code>atoh()</code>	<code>convert.c</code>	Converts an ASCII string representation of a hexadecimal byte into the corresponding hexadecimal byte. For example, the string "a3" is converted to the byte value 0xa3.
<code>atohexint()</code>	<code>convert.c</code>	Converts a four-byte ASCII string representing a two-byte hexadecimal word into the corresponding hexadecimal word. For example, the string "a34b" is converted to the word 0xa34b.
<code>atoi()</code>	<code>convert.c</code>	Converts a string representing a decimal integer into the corresponding integer. This subroutine is from Bilofsky [Ref. 18].
<code>bad_idea_to_record()</code>	<code>expmnt.c</code>	This routine is used in the abridged experiment to prevent <i>record</i> mode from being restarted after a power fault. Without this safeguard, perfectly good data recorded during launch might be erased.

baro_switch()	expmnt.c	Checks to see if the barometric switches have been activated yet. If so, launch must have occurred and an appropriate log entry is made.
bcd_asc()	convert.c	Converts a BCD byte to the corresponding character string representation. For example, 0x17 is converted to "17".
bcd_int()	convert.c	Converts a one-byte BCD number to integer format.
bpageset()	bubble.c	Loads the five parametric registers in the bubble memory controller. Most of these never change. Two, however, do change often. These two specify the page of bubble memory where transfers of data begin. Call this function prior to any operation performing input from or output to the bubble memory to ensure the parameters are correctly specified.
bubcmdmenu()	bubble.c	Displays a menu of low-level bubble memory controller commands. These are useful in testing the bubble memory for proper operation.
bubinit()	bubble.c	Initializes the bubble memory prior to its being used. This initialization must always be done after power is applied and before input and output operations begin.
bubio()	bubble.c	Performs input from and output to the bubble memory.
bubmenu()	bubble.c	Provides a menu of functions permitting the user to perform operations with the bubble memory. These operations include: 1. applying and removing power, 2. initialization, 3. input, 4. output and 5. reading the status of the bubble memory.
bub_off()	bubble.c	Removes power from the bubble memory.
bub_on()	bubble.c	Applies power to the bubble memory.

bubread()	bubrw.s	Takes care of the actual transfer of data from the bubble memory to the controller memory during a read. This routine was written in assembly language in order to achieve a data transfer rate of 16 K bytes per second imposed by the bubble memory hardware.
bubwrite()	bubrw.s	Takes care of the actual transfer of data to the bubble memory from the controller memory during a write. This routine was written in assembly language in order to achieve a data transfer rate of 16 K bytes per second imposed by the bubble memory hardware.
bubxfer()	bubrw.s	Part of the sequence of steps necessary to initialize the bubble memory entails transferring the byte 0xff to the bubble memory 40 times. This routine does this. The routine is written in assembly language for speed, but is called in the same manner as a C routine.
checkprt()	expmnt.c	Checks to see whether or not there is a terminal connected to the RS-232C serial interface port.
clockint()	clock.c	Dates and times in the real time clock are stored in BCD format. This routine converts them to integer format to make it convenient to perform arithmetic with them. Thus, future dates and times can be computed.
clockread()	clock.c	Reads the current date and time from the real time clock.
clockcompare()	clock.c	Compares two clock times to see which one is later than the other.
clockset()	clock.c	Sets the current date and time in the real time clock according to values specified by the user.
clocksum()	clock.c	Adds two dates and times together to produce a new date and time. In practice, one uses this to calculate a future date and time from the current date and time and some desired delay (e.g., 15 minutes).
colder_than()	expmnt.c	Returns the value TRUE if the bubble memory's temperature is below the temperature given in the argument to the function, FALSE otherwise.
ctoh()	convert.c	Converts a single character to its hexadecimal ASCII string representation. For example, 0xa3 is converted to "a3".

delay()	delay.s	Provides a software-driven time delay in increments of 10 ms. Written in assembly language, but used like a C language routine. Adapted from a program by Mr. David Rigmaiden of the Naval Postgraduate School.
display_page0()	expmnt.c	Displays the contents of page 0 in a readable format.
display_data_page()	expmnt.c	Displays the contents of any page in the bubble memory in a readable format. It will not successfully display page 0. Use display_page0() for this purpose.
do_sweep()	expmnt.c	Causes the <i>sweep</i> phase of the experiment to be performed.
dump()	inout.c	Produces a hexadecimal and ASCII dump of any desired region of memory.
dump_clock()	clock.c	Display the date and time on the terminal.
dump_iclock()	clock.c	Display the date and time on the terminal. This differs from dump_clock() in that the dates and times it uses are integers, not Binary Coded Decimal (BCD) numbers.
echo()	inout.c	Sends a single character to the terminal.
expmnt()	expmnt.c	Causes the Vibro-acoustic Experiment to be performed.
fputc()	fputc.c	The UNIWARE compiler provides the standard C output routine printf() to provide output to the standard output device. However, this routine requires the user to provide a routine fputc() to handle the output of a single character to <i>any</i> arbitrary device. We only support output by fputc() to the RS-232C terminal, so this routine is specific to that device. The routine will <i>not</i> output a character if, upon checking, it finds there is no terminal attached to the serial interface port. Thus, when the experiment is operating, calls to printf() are of no effect unless there is a terminal connected.
gethex()	inout.c	Inputs a string representation of a two-digit hexadecimal number from the terminal and converts it to hexadecimal format. For example, "3a" is converted to 0x3a.
gethexint()	inout.c	Gets a four-digit hexadecimal number in string format from the terminal and converts it to a hexadecimal word. For example, "3ab2" is converted to 0x3ab2.

<code>getint()</code>	<code>inout.c</code>	Inputs a string representation of a decimal integer from the terminal and converts it to integer format. For example, "352" is converted to 352.
<code>getpageno()</code>	<code>inout.c</code>	Asks the user for a page number in bubble memory.
<code>get_time()</code>	<code>clock.c</code>	Obtain a valid date and time from the user.
<code>inithardware()</code>	<code>initial.c</code>	Initializes the six ports on NSC810A #1 and #2.
<code>input()</code>	<code>newio.s</code>	Inputs a character from a port. Written in assembly language, but used like a C language routine.
<code>int_bcd()</code>	<code>convert.c</code>	Converts an integer in the range 0 through 99 to BCD format.
<code>issububcmd()</code>	<code>bubble.c</code>	Issues commands to the bubble memory controller and analyzes the status codes which result. In many cases, it will attempt to issue a command repeatedly if there is some failure, doing this up to a specified number of times. This routine is written in C and is not fast enough to handle the read and write commands. Use <code>bubread()</code> and <code>bubwrite()</code> for these.
<code>itoa()</code>	<code>convert.c</code>	Converts an integer to an ASCII string representation. This subroutine is from Bilofsky [Ref. 18].
<code>listen()</code>	<code>expmnt.c</code>	Listens for the Auxiliary Power Units (APUs) to be activated. It also monitors the Vibration-activated Launch Detector and the Barometric Pressure Switches to see if a launch has occurred without detection of the activation of the APUs.
<code>logevent()</code>	<code>expmnt.c</code>	Makes entries into the event log stored in the bubble memory.
<code>log_menu()</code>	<code>expmnt.c</code>	Displays a menu to provide for conveniently changing the contents of bubble memory. This is essential for properly initiating the experiment.
<code>look_ahead()</code>	<code>inout.c</code>	This program can see whether a character has been input from the keyboard without disturbing the input buffer.
<code>main()</code>	<code>main.c</code>	First C language subroutine to get control after start-up. Decides whether to invoke the menu-driven diagnostic routines or to run the Vibro-acoustic Experiment.

mbrk()	mbrk.s	Implements the C language standard library function mbrk() . This function was provided with the Uniware C Compiler.
memory_dump()	main.c	Asks the user for an address in memory and the number of bytes he wants to see displayed. It then provides a hexadecimal and ASCII display of the contents of the selected area of memory on the terminal.
menu()	main.c	Displays the first in a hierarchy of menus permitting the user to test subsystems of the Vibro-acoustic Experiment individually.
monitor_heaters()	expmnt.c	Operates the heaters if the temperature of the bubble memory is too cold. If the temperature is too hot, it shuts the heaters off. Otherwise it leaves the heaters alone.
output()	newio.s	Outputs a byte to a port. Written in assembly language, but used like a C language routine.
portdump()	inout.c	Outputs a string to the terminal.
post_launch()	expmnt.c	Conducts routine monitoring of events upon the completion of the Vibro-acoustic Experiment. These functions continue until the Space Shuttle returns to earth, or until the 10V bus no longer carries sufficient voltage for safe operation of the bubble memories. In the latter case, the experiment stops all operations.
power_status()	power.c	Inputs the one-byte status code from the power relay subsystem.
power_write()	power.c	Sends a one-byte command code to the power relay subsystem.
pwrcnt()	power.c	A menu program which let's the user read the status code from the power relay subsystem or send commands to it.
rdstatreg()	bubble.c	Reads the status register of the bubble memory controller.
record()	expmnt.c	Performs the <i>record</i> phase of the abridged experiment.
rtc()	clock.c	A menu routine allowing the user to set or read the clock, and to test the time-out feature (see testtimeout() in this table).
short_experiment()	expmnt.c	Performs the abridged Vibro-acoustic Experiment.

showbubbuff()	bubble.c	A buffer exists in the controller's memory to hold a copy of data transferred to or from the bubble memory. This routine displays the contents of that buffer either in ASCII characters or hexadecimal.
show_event()	expmnt.c	Converts an event code into an intelligible message which it then displays on the terminal.
show_waketime()	clock.c	Displays the date and time when a time-out will end.
shut_down()	expmnt.c	Removes power from any subsystems which presently have power. It makes a log entry for each such case.
shut_down_no_log()	expmnt.c	Removes power from any subsystems which presently have power. It makes no log entry of its actions.
ssdrmode()	expmnt.c	Issues commands to the Solid State Data Recorder (SSDR) to enter various modes of operation.
ssdr_status()	expmnt.c	Obtains the status code from the Solid State Data Recorder (SSDR).
termin()	inout.c	Inputs a single character from the terminal.
testinput()	inout.c	Asks the user for a hexadecimal port address, reads that port and displays the data read from that port.
testio()	main.c	This routine permits the user to perform input from and output to any port in the system. By "port" we mean here an address in the input and output space.
testoutput()	inout.c	Asks the user for a hexadecimal port address and a hexadecimal byte to be sent to the port, and sends it there.
testpattern()	bubble.c	A buffer exists in the controller's memory to hold a copy of data transferred to or from the bubble memory. This routine permits the user to modify the contents of that buffer.
testtimeout()	clock.c	Lets the user test the time-out feature. For example, he can request a delay of 15 seconds. During this delay, the control program will not respond to input. At the end of this period, it will display the current date and time.

timeout()	clock.c	In one mode of operation, this function computes a "wake-up" time based on the current time and a specified delay. In another mode, it checks to see if a "wake-up" time computed earlier has arrived or not.
tolower()	convert.c	Converts upper case characters to lower case. Non-alphabetic characters are not changed. This subroutine is from Bilofsky [Ref. 18].
uitoh()	convert.c	Converts an unsigned integer to the corresponding hexadecimal ASCII string representation. For example, 45 is converted to "2D"
version()	version.c	Displays the current version number of the control program on the terminal.
voltages_low()	expmnt.c	Checks the 10V bus. If the voltage has fallen too low, this function returns the value TRUE; otherwise it returns the value FALSE.
we_launched()	expmnt.c	Checks for any indications of a launch. These can come from the Vibration-activated Launch Detector or from the Barometric Pressure Switches.

APPENDIX F. SUBROUTINES, IN ALPHABETICAL ORDER WITHIN EACH MODULE

Table 9. MS/DOS FILE INDEX: This table shows the names of the files in the MS DOS files. Subroutines are listed alphabetically by name within each file group.

SOURCE FILE	SUBROUTINE	PURPOSE
bubble.c	bpageset()	Loads the five parametric registers in the bubble memory controller. Most of these never change. Two, however, do change often. These two specify the page of bubble memory where transfers of data begin. Call this function prior to any operation performing input from or output to the bubble memory to ensure the parameters are correctly specified.
bubble.c	bubcmdmenu()	Displays a menu of low-level bubble memory controller commands. These are useful in testing the bubble memory for proper operation.
bubble.c	bubinit()	Initializes the bubble memory prior to its being used. This initialization must always be done after power is applied and before input and output operations begin.
bubble.c	bubio()	Performs input from and output to the bubble memory.
bubble.c	bubmenu()	Provides a menu of functions permitting the user to perform operations with the bubble memory. These operations include: 1. applying and removing power, 2. initialization, 3. input, 4. output and 5. reading the status of the bubble memory.
bubble.c	bub_off()	Removes power from the bubble memory.
bubble.c	bub_on()	Applies power to the bubble memory.

bubble.c	issububcmd()	Issues commands to the bubble memory controller and analyzes the status codes which result. In many cases, it will attempt to issue a command repeatedly if there is some failure, doing this up to a specified number of times. This routine is written in C and is not fast enough to handle the read and write commands. Use bubread() and bubwrite() for these.
bubble.c	rdstatreg()	Reads the status register of the bubble memory controller.
bubble.c	showbubbuff()	A buffer exists in the controller's memory to hold a copy of data transferred to or from the bubble memory. This routine displays the contents of that buffer either in ASCII characters or hexadecimal.
bubble.c	testpattern()	A buffer exists in the controller's memory to hold a copy of data transferred to or from the bubble memory. This routine permits the user to modify the contents of that buffer.
bubrw.s	bubread()	Takes care of the actual transfer of data from the bubble memory to the controller memory during a read. This routine was written in assembly language in order to achieve a data transfer rate of 16 K bytes per second imposed by the bubble memory hardware.
bubrw.s	bubwrite()	Takes care of the actual transfer of data to the bubble memory from the controller memory during a write. This routine was written in assembly language in order to achieve a data transfer rate of 16 K bytes per second imposed by the bubble memory hardware.
bubrw.s	bubxfer()	Part of the sequence of steps necessary to initialize the bubble memory entails transferring the byte 0xff to the bubble memory 40 times. This routine does this. The routine is written in assembly language for speed, but is called in the same manner as a C routine.
clock.c	clockcompare()	Compares two clock times to see which one is later than the other.
clock.c	clockint()	Dates and times in the real time clock are stored in BCD format. This routine converts them to integer format to make it convenient to perform arithmetic with them. Thus, future dates and times can be computed.

clock.c	clockread()	Reads the current date and time from the real time clock.
clock.c	clockset()	Sets the current date and time in the real time clock according to values specified by the user.
clock.c	clocksum()	Adds two dates and times together to produce a new date and time. In practice, one uses this to calculate a future date and time from the current date and time and some desired delay (e.g., 15 minutes).
clock.c	dump_clock()	Display the date and time on the terminal.
clock.c	dump_icklock()	Display the date and time on the terminal. This differs from dump_clock() in that the dates and times it uses are integers, not Binary Coded Decimal (BCD) numbers.
clock.c	get_time()	Obtain a valid date and time from the user.
clock.c	rtc()	A menu routine allowing the user to set or read the clock, and to test the time-out feature (see testtimeout() in this table).
clock.c	show_waketime()	Displays the date and time when a time-out will end.
clock.c	testtimeout()	Lets the user test the time-out feature. For example, he can request a delay of 15 seconds. During this delay, the control program will not respond to input. At the end of this period, it will display the current date and time.
clock.c	timeout()	In one mode of operation, this function computes a "wake-up" time based on the current time and a specified delay. In another mode, it checks to see if a "wake-up" time computed earlier has arrived or not.
convert.c	atoh()	Converts an ASCII string representation of a hexadecimal byte into the corresponding hexadecimal byte. For example, the string "a3" is converted to the byte value 0xa3.
convert.c	atohexint()	Converts a four-byte ASCII string representing a two-byte hexadecimal word into the corresponding hexadecimal word. For example, the string "a34b" is converted to the word 0xa34b.
convert.c	atoi()	Converts a string representing a decimal integer into the corresponding integer. This subroutine is from Bilofsky [Ref. 18].

convert.c	bcd_asc()	Converts a BCD byte to the corresponding character string representation. For example, 0x17 is converted to "17".
convert.c	bcd_int()	Converts a one-byte BCD number to integer format.
convert.c	ctoh()	Converts a single character to its hexadecimal ASCII string representation. For example, 0xa3 is converted to "a3".
convert.c	int_bcd()	Converts an integer in the range 0 through 99 to BCD format.
convert.c	itoa()	Converts an integer to an ASCII string representation. This subroutine is from Bilofsky [Ref. 18].
convert.c	tolower()	Converts upper case characters to lower case. Non-alphabetic characters are not changed. This subroutine is from Bilofsky [Ref. 18].
convert.c	uitoh()	Converts an unsigned integer to the corresponding hexadecimal ASCII string representation. For example, 45 is converted to "2D".
delay.s	delay()	Provides a software-driven time delay in increments of 10 ms. Written in assembly language, but used like a C language routine. Adapted from a program by Mr. David Rigmaiden of the Naval Postgraduate School.
expmnt.c	ad_read()	Gets a character of data from the Analog-to-digital (A/D) Converter.
expmnt.c	adtoint()	Converts a character of raw data from the Analog-to-digital (A/D) Converter into an integer format with the more meaningful units of volts or degrees kelvin.
expmnt.c	alter_page0()	Permits the user to alter the contents of page 0 of the bubble memory. This is required in initializing the experiment.
expmnt.c	bad_idea_to_record()	This routine is used in the abridged experiment to prevent <i>record</i> mode from being restarted after a power fault. Without this safeguard, perfectly good data recorded during launch might be erased.
expmnt.c	baro_switch()	Checks to see if the barometric switches have been activated yet. If so, launch must have occurred and an appropriate log entry is made.

expmnt.c	checkprt()	Checks to see whether or not there is a terminal connected to the RS-232C serial interface port.
expmnt.c	colder_than()	Returns the value TRUE if the bubble memory's temperature is below the temperature given in the argument to the function, FALSE otherwise.
expmnt.c	display_data_page()	Displays the contents of any page in the bubble memory in a readable format. It will not successfully display page 0. Use display_page0() for this purpose.
expmnt.c	display_page0()	Displays the contents of page 0 in a readable format.
expmnt.c	do_sweep()	Causes the <i>sweep</i> phase of the experiment to be performed.
expmnt.c	expmnt()	Causes the Vibro-acoustic Experiment to be performed.
expmnt.c	listen()	Listens for the Auxiliary Power Units (APUs) to be activated. It also monitors the Vibration-activated Launch Detector and the Barometric Pressure Switches to see if a launch has occurred without detection of the activation of the APUs.
expmnt.c	logevent()	Makes entries into the event log stored in the bubble memory.
expmnt.c	log_menu()	Displays a menu to provide for conveniently changing the contents of bubble memory. This is essential for properly initiating the experiment.
expmnt.c	monitor_heaters()	Operates the heaters if the temperature of the bubble memory is too cold. If the temperature is too hot, it shuts the heaters off. Otherwise it leaves the heaters alone.
expmnt.c	post_launch()	Conducts routine monitoring of events upon the completion of the Vibro-acoustic Experiment. These functions continue until the Space Shuttle returns to earth, or until the 10V bus no longer carries sufficient voltage for safe operation of the bubble memories. In the latter case, the experiment stops all operations.
expmnt.c	record()	Performs the <i>record</i> phase of the abridged experiment.
expmnt.c	short_experiment()	Performs the abridged Vibro-acoustic Experiment.

expmnt.c	show_event()	Converts an event code into an intelligible message which it then displays on the terminal.
expmnt.c	shut_down()	Removes power from any subsystems which presently have power. It makes a log entry for each such case.
expmnt.c	shut_down_no_log()	Removes power from any subsystems which presently have power. It makes no log entry of its actions.
expmnt.c	ssdrmode()	Issues commands to the Solid State Data Recorder (SSDR) to enter various modes of operation.
expmnt.c	ssdr_status()	Obtains the status code from the Solid State Data Recorder (SSDR).
expmnt.c	voltages_low()	Checks the 10V bus. If the voltage has fallen too low, this function returns the value TRUE; otherwise it returns the value FALSE.
expmnt.c	we_launched()	Checks for any indications of a launch. These can come from the Vibration-activated Launch Detector or from the Barometric Pressure Switches.
fputc.c	fputc()	The UNIWARE compiler provides the standard C output routine printf() to provide output to the standard output device. However, this routine requires the user to provide a routine fputc() to handle the output of a single character to <i>any</i> arbitrary device. We only support output by fputc() to the RS-232C terminal, so this routine is specific to that device. The routine will <i>not</i> output a character if, upon checking, it finds there is no terminal attached to the serial interface port. Thus, when the experiment is operating, calls to printf() are of no effect unless there is a terminal connected.
initial.c	inithardware()	Initializes the six ports on NSC810A #1 and #2.
inout.c	allow_ctrl_interrupts()	Processes the special characters CTRL S and CTRL Y when read from the keyboard. CTRL S is a toggle switch. The first time it is pressed, the display halts. The second time it is pressed, the display resumes. Subsequently its function alternates between these two. CTRL Y invokes the diagnostic subsystem.

inout.c	dump()	Produces a hexadecimal and ASCII dump of any desired region of memory.
inout.c	echo()	Sends a single character to the terminal.
inout.c	gethex()	Inputs a string representation of a two-digit hexadecimal number from the terminal and converts it to hexadecimal format. For example, "3a" is converted to 0x3a.
inout.c	gethexint()	Gets a four-digit hexadecimal number in string format from the terminal and converts it to a hexadecimal word. For example, "3ab2" is converted to 0x3ab2.
inout.c	getint()	Inputs a string representation of a decimal integer from the terminal and converts it to integer format. For example, "352" is converted to 352.
inout.c	getpageno()	Asks the user for a page number in bubble memory.
inout.c	look_ahead()	This program can see whether a character has been input from the keyboard without disturbing the input buffer.
inout.c	portdump()	Outputs a string to the terminal. interface port.
inout.c	termin()	Inputs a single character from the terminal.
inout.c	testinput()	Asks the user for a hexadecimal port address, reads that port and displays the data read from that port.
inout.c	testoutput()	Asks the user for a hexadecimal port address and a hexadecimal byte to be sent to the port, and sends it there.
main.c	main()	First C language subroutine to get control after start-up. Decides whether to invoke the menu-driven diagnostic routines or to run The Vibro-acoustic Experiment.
mbrk.s	mbrk()	Implements the C language standard library function <code>mbrk()</code> . This function was provided with the Uniware C Compiler.
main.c	memory_dump()	Asks the user for an address in memory and the number of bytes he wants to see displayed. It then provides a hexadecimal and ASCII display of the contents of the selected area of memory on the terminal.
main.c	menu()	Displays the first in a hierarchy of menus permitting the user to test subsystems of the Vibro-acoustic Experiment individually.

main.c	testio()	This routine permits the user to perform input from and output to any port in the system. By "port" we mean here an address in the input and output space.
newio.s	input()	Inputs a character from a port. Written in assembly language, but used like a C language routine.
newio.s	output()	Outputs a byte to a port. Written in assembly language, but used like a C language routine.
power.c	power_status()	Inputs the one-byte status code from the power relay subsystem.
power.c	power_write()	Sends a one-byte command code to the power relay subsystem.
power.c	powercnt()	A menu program which let's the user read the status code from the power relay subsystem or send commands to it.
version.c	version()	Displays the current version number of the control program on the terminal.

APPENDIX G. CONTROL PROGRAM DOCUMENTATION

We presented a general description of the software as a whole in Chapter IV. DESIGN OF THE CONTROL SOFTWARE on page 43. This included a moderately detailed description of the flowcharts which describe the system, beginning with Flowchart 0 in Figure 20 on page 48. This appendix contains more detailed descriptions of the operation of each subroutine in the control program. A basic knowledge of the C programming language is assumed.

We have grouped the functions into two broad categories:

1. major subroutines, and
2. support subroutines.

The descriptions in this appendix are best understood by referring to the source code in APPENDIX H. CONTROL PROGRAM SOURCE CODE on page 150.

In Section A. Major Subroutines and Functions on page 108 we present the major subroutines and functions of the control program in an order based roughly on their position in the hierarchy of function calls. This section therefore follows the overall structure of the control program.

Referring again to Flowchart 0 in Figure 20 on page 48, we see that the control program contains two major parts:

1. One performs the Vibro-acoustic Experiment.
2. The other operates a menu-driven system to permit testing of the system on the ground.

Once we have discussed the major subroutines, there will remain numerous lesser subroutines which we describe in Section B. Supporting Subroutines and Functions on page 121. We provide two tables to make it easier quickly to ascertain the purpose of subroutines and their locations in several different source files. Table 8 on page 91 lists all subroutines by name, and shows in which MS DOS source files subroutines are located. Table 9 on page 99 lists the contents of each MS DOS source file in alphabetic order by name.

In general, the program attempts to display many diagnostic messages on the terminal using the `printf()` function. This function was supplied with the C compiler, but it in turn calls a function called `fputc()` *not* supplied with the compiler. The purpose of

the subroutine `fputc()` is to accept a character from the `printf()` function and to send it to the terminal for display. We created this subroutine, and its description is contained in Section B. Supporting Subroutines and Functions on page 121. This function always checks to see whether there is a terminal attached or not. If not, it makes no attempt to display any messages on the terminal. Henceforth, whenever we say that something will appear on the terminal, it will be understood that this will only occur if the terminal is attached.

A. MAJOR SUBROUTINES AND FUNCTIONS

1. `main()`

This is the beginning point for any C language program. It is called by the start-up code, which is written in Z-80 assembly language. The `main()` program first initializes pointers to the buffers which will hold data from the bubble memory. There are two formats for such data. One is used in page zero of the memory, which is used to record the current status of the experiment. The other format is used in all other pages of the bubble memory to record all actions and measurements taken during the experiment. The buffers are treated both as arrays and as structures. When they are treated as arrays, it is easy to transfer the data to or from the bubble memory. When they are treated as structures, it is easy to extract individual fields of data. By forcing the pointers `pagezero` and `log_page` to point to the arrays `page0_buffer` and `log_buffer` respectively, we can access the data subsequently by using either the pointer to the structure or the name of the array as appropriate.

The `main()` program then calls `inithardware()` to initialize the two NSC810A RAM-I/O Timer chips on the controller board. Next it checks to see if there is a terminal attached by calling `checkprt()`. The absence of a terminal implies that the apparatus is now installed in the Space Shuttle and the controller should therefore perform the experiment. Therefore, if there is no terminal attached, `main()` will call `expmnt()`, which performs the Vibro-acoustic Experiment.

If there is indeed a terminal attached, `main()` calls `shut_down_no_log()`, whose function is to remove power from all subsystems without logging that action in the bubble memory. The reason for removing power is to ensure that all the subsystems are in a known state at the outset. The reason for not wishing to log this action is that the log entries should only be made during the course of the experiment. Since the controller is about to enter the menu subsystem, it is not going to perform the experiment and so no log entry is appropriate.

Next `main()` calls `menu()`, from which all other testable sections of the control program can be selected. The option `EXPERIMENTOK` permits the menu diagnostic subsystem to invoke the program `expmint()` later, if the user wishes to do so. This would permit him to perform the experiment on the ground and so test its operation.

2. `void inithardware(void)`

This subroutine initializes the two NSC810A RAM-I/O-Timer chips on the controller board. The uses of the pins of port A in each chip are given in Table 4 on page 17 and Table 5 on page 17; those of Port B are given in Table 2 on page 15 and Table 3 on page 16; and those of port C are given in Table 6 on page 18 and Table 7 on page 19.

MDR1 is the Mode Definition Register of the NSC810A #1. Writing a 0x00 to it puts port A₁ into basic I/O mode, which is the simplest method of I/O supported by this chip.²²

DDRA1 is the Data Direction Register of port A₁ of the NSC810A #1. Writing 0xff to it causes each of its bits to be configured for output.

DDRB1 is the Data Direction Register of port B₁ of the NSC810A #1. Writing 0xff to it causes each of its bits to be configured for output.

DDRC1 is the Data Direction Register of port C₁ of the NSC810A #1. Writing 0x30 to it causes bits 0 through 3 and bits 6 and 7 to be configured for input. Bits 4 and 5 are configured for output, although bit 5 is not used in the Vibro-acoustic Experiment. Note: this is only a 6 bit port; bits 6 and 7 do not exist.

TM01 is the register for setting the mode of Timer 0 in NSC810A #1. Writing 0x00 to it will stop the timer, an action which must be performed *before* changing its mode. Writing 0x25 will cause the timer to produce a square wave without "prescaling" and with "single precision". When prescaling is not used, every pair of input clock cycles is used to advance the timer's counter by one. When single precision is selected, only the low byte of the timer will ever be read.

TOLB1 and TOHB1 are the registers for the low byte and high byte respectively of the modulus for Timer 0 in NSC810 #1. This number serves to initiate the timer counter. During subsequent operation, the counter is decremented once every clock period. Each time the counter reaches 0, the timer output switches to the opposite state and the timer is reloaded. We write 0x07 to the low byte and 0x00 to the high byte, so the modulus is 7. This means that after every seven cycles, the clock is reloaded. The

²² With basic I/O, there is no handshaking (see Glossary) with support hardware.

reloading consumes a further cycle, and it takes two complete reloads to go through one cycle of the output. The period thus is $2 \times (7 + 1) = 16$ clock periods. The NSC800 is driven by a $4.9152 \div 2 = 2.4576$ MHz clock. So 16 clock periods take $6.51 \mu\text{s}$, for a clock frequency of $\frac{1}{6.51 \mu\text{s}} = 153.6$ kHz. This signal is used as a baud-rate generator on the controller board; it is fed to an Intersil 6402 UART which further divides the frequency by 16, yielding a 9600 baud transmission rate at which to drive the RS-232C interface.

START01 is the start port of Timer 0 in NSC810 #1. Writing anything to this port causes the newly programmed timer to start operating.

MDR2 is the Mode Definition Register of the NSC810A #2. Writing a 0x00 to it puts port A₂ into basic I/O mode. This is the simplest method of I/O supported by this chip.²³

DDRA2 is the Data Direction Register of port A₂ of the NSC810A #2. Writing 0x00 to it causes each of its bits to be configured for input.

DDRB2 is the Data Direction Register of port B₂ of the NSC810A #2. Writing 0x00 to it causes each of its bits to be configured for input.

DDRC2 is the Data Direction Register of port C₂ of the NSC810A #2. Writing 0x31 to it causes bits 1 through 3 to be configured for input. Bits 0, 4 and 5 are configured for output. Bits 1 and 2 are not in use. Note: this is only a 6 bit port; bits 6 and 7 do not exist.

TM02 is the register for setting the mode of Timer 0 in NSC810A #2. Before you can change the mode, you must first stop the timer. Writing 0x00 to it does this. Writing 0x25 will cause the timer to produce a square wave without "prescaling" and with "single precision". When prescaling is not used, every pair of input clock cycles is used to advance the timer's counter by one. When single precision is selected, only the low byte of the timer will ever be read.

T0LB2 and T0HB2 are the registers for the low byte and high byte respectively of the modulus of Timer 0 in NSC810 #2. This number serves to initiate the timer counter. Once every clock period, the counter is decremented. Each time the counter reaches 0, the timer output switches to the opposite state and the timer is reloaded. We write 0x01 to the low byte and 0x00 to the high byte, so the modulus is 1. This means that after 1 cycle, the clock is reloaded. Now the reloading consumes a further cycle, and it takes two complete reloads to go through one cycle of the output. The

²³ With basic I/O, there is no handshaking (see Glossary) with support hardware.

period thus is $2 \times (1 + 1) = 4$ clock periods. The NSC800 is driven by a $4.9152 \div 2 = 2.4576$ MHz clock. So 4 clock periods take $1.628 \mu\text{s}$, for a clock frequency of $\frac{1}{1.628 \mu\text{s}} = 614.4$ kHz. This frequency is used as a clock for the National Semiconductor ADC0816 Analog-to-digital (A/D) Converter.

When driven by a clock of frequency 640 kHz, the A/Ds normally can complete the conversion of an analog signal to a digital value in around $100 \mu\text{s}$. The frequency we are using here, 614.4 kHz, is close to this, so we should get comparable performance. [Ref. 19: pp. 8-71 to 8-81]

START02 is the start port for Timer 0 in NSC810 #2. Writing anything to this port causes the newly programmed timer to start operating.

Finally, we clear bits 4 and 5 of port C_2 by writing 0x03 to the port C_2 "bit clear" register, BCLRC2. The purpose of this is to ensure that power to the bubble memory remains off, and to ensure that the bubble memory's reset line is held low. Strictly speaking, this should not be necessary, since the registers of the NSC810 are initialized to be zeros. However, we take nothing for granted, and this precaution helps preclude the loss of the bubble memory's contents that might result from an improper application of power.

3. char checkprt(void)

This function inspects the TERMON bit (bit 3) of Port C in NSC810 #2. This bit is a 0 if there is an RS-232C terminal connected to the controller. It is a 1 otherwise. The function returns a TRUE in the first case; a FALSE in the second.

4. void shut_down_no_log(void)

This subroutine removes power from any subsystems which are currently on. It does not record the fact in the bubble memory log, which is the only respect in which it differs from the subroutine `shut_down()`. It obtains a character describing the position of each of the relays in the power subsystem by calling the function `power_status()`. The series of `if` statements which then follows causes successive bits of that character to be tested. Every time one of these bits indicates that a relay is in the 'on' position, that relay is turned off with a call to `power_write()`.

5. char menu(char experiment_flag)

This function is at the top of a hierarchy of diagnostic subroutines. The function calls the sub-function `version()` whose only purpose is to display the number and date of the current version of the control program. It next presents a menu from which the user can select any of a number of categories of diagnostic tests. The function `termin()` is used to obtain a single character from the keyboard, that character is con-

verted to lower case by **tolower()** (if it was not already in lower case), and the character is displayed on the terminal. That character is used by the **switch** to select a **case** statement appropriate to the user's choice. The entire process will be executed repetitively. The only way to leave it is by choosing to run the experiment. If this is done, the function **expmnt()** gets control.

To cause a software reset, the program executes an assembler instruction **jp 0**. This function has the effect of restarting the controller at address 0 of memory. This is the same address at which execution begins when power is first applied. All variables are set to their initial values, other start-up functions are performed as usual, and the program **main()** begins to execute anew.

The function **rtc()** accesses the real-time clock diagnostic subroutines.

The function **pwrcnt()** access the power subsystem diagnostic subroutines.

The function **bubmenu()** accesses the diagnostic subroutines which can be used to test the bubble memory module. The tests available through this selection all are very low-level tests.

When choice E is made, the controller enters a **for** loop and successively reads each of the analog-to-digital (A/D) converter channels by calling the sub-function **adread()**. This function returns an eight-bit number **addata** which is proportional to the value read by the A/D converter. A call to **printf()** displays this number along with a descriptive **adcaption** (defined in the file **global.c**). The first three readings are known to be voltages. The remaining values are temperatures, so they are displayed in a slightly different format. Furthermore, depending on which channel the A/D converter read, the number read may represent different magnitudes in the measured units. For example, the number 102 may represent 4V or 270°K, depending on which channel was read.

Voltages, fall either into the range [0, 10]V or the range [0, 20]V. Temperatures fall into the range [0, 500]°K. The function **adtoint()** converts the value read by the A/D converter into its value in degrees Kelvin or in hundredths of volts, whichever is applicable. The converted value is then displayed using the **printf()** function. To get two converted readings per line, carriage returns are placed at the end of every other displayed value, only.

There are two possibilities if choice F is made. One is that **experiment_flag** is TRUE; the other is that it is FALSE. The former case always occurs when **menu()** is called the first time, from **main()**. However, it is possible to interrupt the execution of the experiment and to enter the menu subsystem recursively. It is not possible to make

menu choice F under these circumstances. To restart the experiment would require first making choice A to reset the system.

The function `testio()` is called when choice G is made. Its purpose is to allow low-level testing of the peripheral devices.

The function `memory_dump()` is called when choice H is made. Its purpose is to display the contents of the controller's memory. This is useful only in debugging the software.

The function `log_menu()` is called when choice I is made. Its purpose is to allow the contents of the bubble memory to be displayed. It differs from the functions called when choice D is made in that the contents of the bubble memory are regarded by `log_menu()` as formatted data areas, not just as collections of characters. This means that the data stored in the bubble memory during execution of the experiment can be displayed in an intelligible format, and the experiment's status, stored in page 0, also can be displayed in a readable format. The function `log_menu()` also allows the status to be modified in order to affect the manner in which the controller performs the experiment. The details of how to do this are contained in Chapter V. HOW TO GET THE EXPERIMENT READY FOR A LAUNCH on page 63.

6. `void version(void)`

This function displays the number and date of the current version of the control program on the terminal.

7. `void rtc(void)`

This function displays a menu of functions related to the operation of the real-time clock. The clock can be read, set or tested from here. The method of displaying the menu, reading the choice, and taking the appropriate action is identical to that used in the function `menu()` described earlier. The function `rtc()` differs only in the choices and actions possible.

Choice A causes the function `clockread()` to be called. It stores the current date and time in a structure whose pointer is `clock`. The function `dump_clock()` is called next; it displays the date and time on the terminal. This choice is provided to verify that the real time clock is working correctly.

Choice B causes the function `clockset()` to be called. It permits the user to set the current date and time. The real time clock is powered by its own battery, so this option should seldom be required.

Choice C causes the function `testtimeout()` to be called. Its purpose is to permit the operation of the timeout feature to be tested. It is useful only in debugging the software.

8. void clockread(struct datetime *your_clock)

This function inputs the binary-coded-decimal (BCD) time from the real-time clock and places the results in a structure pointed to by `your_clock`. If the current number of seconds changes between the start and end of reading, it means that the clock has advanced to a subsequent time. To preclude the reading of an incorrect time, the input sequence is repeated in the hope that an advance will not occur again. This can happen up to $10 \times \text{TRIES}$ times.

For example, suppose the time were 9:59:59 when the seconds and minutes were read. The clock might advance to 10:00:00 before the hours were read. Then the time read would appear to be 10:59:59, which is wrong by one hour. By reading it again, we may avoid this error, but there is no obvious way to guarantee it without stopping the clock. Doing so would be disadvantageous, since it would affect timing relationships in an unpredictable manner, so we chose not to stop the clock but to take our chances and try reading it again.

9. void dump_clock(struct datetime *clock)

This function displays on the terminal the time stored in a structure pointed to by `clock`. To do this it calls the function `bcd_int()`, which converts the BCD values in the date and time provided by the real time clock into decimal equivalents. These converted values are then displayed by the function `printf()`.

10. void clockset(struct datetime *clock)

This function first calls the function `get_time()` to ask the user for the current date and time. The time specified is left in the structure pointed to by `clock`. The function `clockset()` then stores the date and time in the real-time clock by repeated calls to `output()`.

11. void testtimeout(void)

This allows the user to test that the time-out function is working. The time-out function enables the control program to continue normal processing while waiting for some amount of time to elapse.

For example, after launch the controller will monitor the Solid State Date Recorder (SSDR) for completion of recording. However, it will also initialize a time-out of three minutes, and will stop waiting for the SSDR if this time should elapse before the SSDR signals completion. The `testtimeout()` function allows the user to test the time-out

feature for any number of seconds, minutes or hours. A menu is presented to the user using the same method already outlined in the description of the function `menu()`. The units of the specified delay depends on the menu choice made. The function `getint()` is called to obtain the number of units of delay that the user wants. The current time then is obtained with a call to `clockread()`, and it is displayed on the terminal with a call to `dump_clock()`. The `timeout()` function then is called to initialize the delay according to the number of delay units specified by the user. A `while` loop calls `timeout()` repeatedly with the `NULL` parameter. This parameter causes the `timeout()` function to check to see if the desired wake-up time has arrived or not. As long as it has not yet arrived, that function returns `FALSE` and the program continues to loop. If other statements were provided before the end of the loop, then they would be performed repeatedly until the function `timeout()` finally returned `TRUE`, signifying that the desired amount of time had elapsed. The function `testtimeout()` has no such instructions, but when the delay period is over, it rings the bell and once again reads and displays the current time.

12. `void pwrcnt(void)`

This function displays a menu to allow the user to test the operation of the power board relays. Any of the attached units, such as the SSDR, can be switched on or off from this menu. The method of displaying the menu is the same as that already given in the description of the function `menu()`. Any menu choice from A through J is converted to a number in the range [0,9] by subtracting the character 'a' from it. This number is then used as an index into array `relay` to select the command to be issued to the power control subsystem through a call to the function `power_write()`. Choice K causes the power subsystem's status to be read with a call to `power_status()` and then displayed on the terminal. The meaning of this byte is shown in Table 2 on page 15.

13. `void bubmenu(void)`

This function displays a menu which lets the user test the bubble memory on the controller circuit board. The method of displaying the menu is the same as that already given in the description of the function `menu()`.

Choice A causes a call to `bub_on()`.

Choice B causes a call to `bub_off()`.

Choice C attempts to initialize the bubble memory with a call to `bubinit()`. The results of this attempt are then explained with a call to `printf()`.

Choice D causes a call to `bubcmdmenu()`.

Choice E causes a call to **testpattern()**. The character string **tempbuffer** is provided to this function for storage of a string of characters entered by the user from the keyboard.

Choice F causes the contents of **tempbuffer** to be displayed in ASCII format.

Choice G causes the contents of **tempbuffer** to be displayed in hexadecimal format. This would be useful if the buffer had been loaded from the bubble memory and if it contained unprintable characters. Such would be the case if the contents of the bubble memory had been generated by performing the experiment, since the experiment formats the data in characters which may not all be capable of being displayed.

Choice H calls **getpageno()** to ask the user which page of the bubble memory he wishes to access. It then calls **bubio()** to transfer the contents of the buffer into that page of the bubble memory.

Choice I calls **getpageno()** to ask the user which page of the bubble memory he wishes to access. It then calls **bubio()** to transfer the contents of that page of the bubble memory into the buffer.

Choice J causes a call to **rdstatreg()**, which reads and displays the contents of the bubble memory controller's status register. The format of this register is discussed in detail in [Ref. 1].

14. **char bub_on(void)**

This function applies power to the bubble memory on the controller circuit board.

15. **void bub_off(void)**

This function removes power from the bubble memory on the controller circuit board.

16. **char bubinit(void)**

This subroutine initiates the bubble memory on the controller circuit board. Power must have been applied first. The sequence of commands is described in [Ref. 1: pp. 38-39b]. It is as follows:

1. Issue the **BABORT** (abort) command to the bubble memory.
2. Set up the parametric registers, pointing to page 0 of the bubble memory.
3. Issue the **BINIT** (bubble initialize) command.
4. Issue the **BFIFORESET** (bubble FIFO reset) command to reset the first-in, first-out (FIFO) buffer in the controller's bubble memory.
5. Transfer 40 0xff characters to the FIFO buffer in the bubble memory.

6. Issue the BWRBLREG (bubble write boot loop register) command. At this point, the bubble memory is ready for reading and writing.

17. void bubcmdmenu(void)

This subroutine allows the user to issue any of the following commands to the bubble memory, one at a time:

1. Abort
2. Load parametric registers
3. Initialize
4. Reset the FIFO buffer
5. Perform the transfer of 40 0xff characters to the FIFO
6. Write the boot loop register

These commands are issued by `bubinit()`, but are provided separately here to permit detailed testing of the initialization process.

18. void testpattern(char buffer[])

This subroutine permits the user to fill a buffer in RAM with characters to be written to the bubble memory. Up to `PAGELENGTH` characters can be written at a time. Its purpose is to enable the user to verify that data can be written to the bubble memory and read back successfully later.

This subroutine begins by placing a 0 in the variable `c`. It asks the user to enter a string of characters from the keyboard, and then enters a loop. It will continue reading up to `PAGELENGTH` characters. If it encounters a carriage return, it will place blanks in the remainder of the buffer.

19. void showbubbuff(char buffer[], char mode)

This subroutine will display the contents of `buffer` either in ASCII format or in hexadecimal representation, according to the value of `mode`. This parameter can be either ASCII or HEX. The ASCII format would be suitable if it were known that the bubble memory page buffer contained only printable characters, as it would if it had been filled by `testpattern()`. The hexadecimal format would be suitable if it were known that the bubble memory page previously read contained unprintable characters, or if the contents were unknown.²⁴

²⁴ It may be unwise to risk sending potentially unprintable characters to the terminal, since some of them have surprising results, such as clearing the screen.

20. `char bubio(char command, int page, char *buffer)`

This subroutine permits reading from or writing to any page of the bubble memory. Pages can fall in the range 0 through 8191. Commands can be either one of BREAD or BWRITE. The data is placed into or read from the buffer pointed to by `buffer`.

To operate the bubble memory when the temperature falls below 10°C may cause its contents to be destroyed. A call to the function `colder_than()` precludes this from happening. If that function returns TRUE, then it must be too cold. The function `bubio` returns a FALSE to indicate that it was unsuccessful in accessing the bubble memory.

To minimize power consumption, the subroutine applies power to the bubble memory before the operation begins and removes it again at the end of the transfer. It calls `bpageset()` to set the parametric registers so as to allow the correct page of bubble memory to be transferred. It then calls `bubread()` or `bubwrite()` as appropriate. After the transfer is completed, the subroutine reads the bubble status register to see if the operation was successful or not. The `bubio()` subroutine returns a TRUE if the transfer worked; FALSE otherwise.

21. `void rdstatreg(void)`

This subroutine lets the user check the contents of the bubble memory status register. The meaning of its contents is shown in Table 10 on page 119. To obtain the status code, this subroutine calls the function `input()`, which reads the contents of the port BUBCTRL (port 0x41). This port yields the status code, which is then converted to hexadecimal format using the function `ctoh()` and is displayed.

22. `void expmnt(void)`

This function performs the experiment. Its first task is to call `initialize()`. This subroutine retrieves the current mission status from page 0 of the bubble memory. If there is no more room in the bubble memory, a value of FALSE will be returned. Although the experiment will be performed, no entries can be made in the log. The Solid State Data Recorder (SSDR) may therefore still be able to record acoustic data successfully. There will be no log of the events as they occur, however.

The function `expmnt()` next checks to see whether the flag `full_experiment` in page 0 is TRUE or FALSE. If not, the function `short_experiment()` is called to perform the abridged experiment. Otherwise, the unabridged experiment is to be performed by `expmnt()`.

Table 10. BIT ASSIGNMENTS FOR THE BUBBLE MEMORY CONTROLLER (BMC) STATUS BYTE: From [Ref. 1 : Chapter 3, p. 12].

Bit	Value	Meaning
0	1	FIFO Ready. The FIFO buffer is ready to transfer data.
	0	The FIFO buffer is not ready.
1	1	Parity error.
	0	No parity error.
2	1	Uncorrectable error.
	0	No uncorrectable error.
3	1	Timing error.
	0	No timing error.
4	1	OP FAIL. The current operation failed.
	0	No OP FAIL.
5	1	OP COMPLETE. The current operation is complete.
	0	No OP COMPLETE.
6	1	Busy. This means that a command has been accepted but is not yet complete. The BMC stays busy throughout a data transfer.
	0	Not busy.

The next step is to initiate the *sweep* phase, if this has not already been done. Recall that this might have occurred if power had been removed from the controller at an earlier time, whether by human intervention or through a fault. If the *sweep* phase is required, the function `do_sweep()` is called to do it.

Next the controller must decide whether or not a launch has already occurred. It consults the `launchdone` flag in page 0 of the bubble memory. If this flag is `TRUE`, the Space Shuttle launched earlier. Otherwise, we must listen for the activation of the Auxiliary Power Units (APUs) by calling the function `listen()`. When this function completes its job, it will return a mission status. This can take on any one of the following values:

DAPUON

The activation of the APUs has been detected.

DLAUNCH

The activation of the APU's was never detected, but launch was detected. This may be the case if the Vibration-activated Launch Detector detects the vibration associated with the ignition of the solid rocket motors or if the Barometric Pressure Switches detect an ascent.

DUSERNOAPU

The system is being tested on the ground and the user depressed a key while the system was listening for the APU's. This provides a means of terminating the period of waiting for a signal.

If `listen()` detects anything, then the function `expmnt()` will turn on the Analog-to-Digital (A/D) Converter by sending the code `ADON` to the function `power_write()`. It then will turn on the Solid State Data Recorder (SSDR) by sending the code `SSDRON` to the same function. Both these actions will be logged in the bubble memory by the function `logevent()`. If `listen()` had returned the mission status code `DAPUON`, then `expmnt()` commands the SSDR to enter *scroll* mode, which means that it will start recording ambient noise. Since the APU's are now on, we know that a launch must occur within seven minutes, or the mission will be scrubbed by NASA. We want to wait at least this long. To be on the conservative side, we begin a ten minute time-out period, during which we wait for some indication of a launch. The function `we_launched()` will return the mission status code `DLAUNCH` if it detects such an indication. The function `look_ahead_discard()` checks to see whether, during ground testing, the user has depressed a key during this time-out period. If so, we regard the time-out as having been completed. This permits accelerated testing of the system without always waiting for the end of the full time-out period. Eventually one of the two conditions will have occurred and the waiting period will end.

If the launch had occurred at some earlier time, we would end up in the next section of the code in `expmnt()`. The fact that a launch had occurred previously would be logged by calling `logevent()` with the argument `PRIORLAUNCH`, and the mission status would be set to this same value.

The next section of code is executed only if a launch is in progress. The SSDR is commanded to leave *scroll* mode and enter *launch* mode. The SSDR has only enough memory to record two minutes of noise after a launch. We initiate a three-minute time-out period so that if the SSDR fails to report completion, we will still be able to go on to other tasks. During the period of this time-out, we want to ensure that a launch is recorded in page 0 of the bubble memory, if in fact a launch has occurred. If the `launchdone` flag in page 0 has not been made `TRUE` yet, `expmnt()` calls `baro_switch()`. This function will check the condition of the barometric switches. If either

one has fired, it will make the **launchdone** flag TRUE. The barometric switches are regarded as the only thoroughly reliable indication of a launch.

We will terminate the *launch* phase either because the SSDR reports completion or because the time-out has occurred. We record whichever of these is the case by calling **logevent()** with either the argument **DOPCOMP** or **DNOOPCOMP**, respectively.

Unless **expmnt()** detected that the launch had been aborted, the experiment will next invoke the function **post_launch()**. This function will keep control until power is removed from the experimental apparatus.

B. SUPPORTING SUBROUTINES AND FUNCTIONS

The major modules of the control program were described in Section A. Major Subroutines and Functions on page 108. Subroutines not described there are described here. They are listed alphabetically by the name of the source file in which they are defined, and alphabetically by function name within file name.

1. File **bubble.c**

a. *void bpageset(int page)*

This subroutine initializes the parametric registers in the bubble memory. There are five of these, and they contain information about the bubble memory's status and about upcoming data transfers. The meaning of the fields within these registers is given in Table 11 on page 122. A complete description of their use is given in [Ref. 1: pp. 7-12], from which the information in Table 11 on page 122 is taken.

Table 11. CONTENTS OF THE PARAMETRIC REGISTERS IN THE BUBBLE MEMORY CONTROLLER: Extracted from [Ref. 1 : Chapter 3, pp. 7-12].

REGISTER ADDRESS	REGISTER NAME	BIT FIELD	CONTENTS
0x0b	Least Significant Byte of the Block Length Register	0-7	Least significant eight bits of the block length. The block length is the number of pages transferred to or from the bubble memory at one time.
0x0c	Most Significant Byte of the Block Length Register	0-2	Most significant three bits of the block length. Thus there are 11 bits in the block length, permitting up to $2^{11} = 2048$ pages to be transferred at once.
		3	Unused
		4-7	The number of Formatter Sense Amplifier channels available. The binary value 0001 is appropriate here because we have only one bubble memory attached. Two channels are used to communicate with the bubble memory. With a single bubble memory available, the page size is defined to be 64 bytes in length.

0x0d	Enable Register	0	Interrupt enable (normal). We set this to 0 because we are not using interrupts to communicate with the bubble memory controller.
		1	Interrupt enable (error). We set this to 0 because we are not using interrupts to communicate with the bubble memory controller.
		2	Direct Memory Access (DMA) Enable. We set this to 0 because we are not using DMA with the bubble memory.
		3	Reserved by Intel.
		4	Write Bootloop Enable. The bootloop is used internally to the bubble memory. It need never be rewritten except as part of a diagnostic test. We let this be 0 since we don't want to modify the bootloop.
		5	Enable Read Corrected Data (RCD). We set this to 1 to permit the format sense amplifier to apply error correction to erroneous data. If the error is uncorrectable, then the erroneous data will be transferred to the host processor.
		6	Enable Internally Corrected Data (ICD). Setting this causes the bubble memory to notify the host processor of its inability to correct erroneous data. In this case, it does not transfer that data. We set this to 0 and don't use the feature.
0x0e	The Least Significant Byte of the Address Register	7	Enable Parity Interrupt. We set this bit to 0 because we are not using interrupts.
		0-7	The least significant byte of the address. The address refers to the particular page within the bubble memory where data transfers are to begin. Since we are using a block length of one page, this actually addresses the single page we are transferring.

0x0f	Most Significant Byte of the Address Register	0-4	Most significant five bits of the address. Thus there are $2^{13} = 8192$ pages in the bubble memory.
		5-7	Magnetic Bubble Memory (MBM) Select. This field controls which bubble memory is addressed. Since we only are using one bubble memory, we set this to all zeroes.

b. char issububcmd(char command)

This subroutine is used to issue a command to the bubble memory controller on the main controller circuit board. The sequence it follows is given in detail in [Ref. 1: pp.40-45]. For our purposes, the sequence is as follows:

1. Make sure the BUSY bit is 0 before sending any command (except ABORT). To do this, we read the status code in the BUBCTRL port and check the BBUSY bit. When this is a 0, we can proceed. More than one attempt will be made to succeed in this. If the check fails repeatedly, the subroutine displays the status code and returns a value of FALSE.
2. Issue the command to the bubble memory controller by calling the function `output()`.
3. Check to see that the command was accepted. This is signalled by the bubble memory controller's setting the BUSY bit once again. If the BUSY bit is not set within a reasonable amount of time, the command was not accepted. In the case of the commands FIFO RESET and WRITE BOOTLOOP REGISTER, we can ignore the fact that the BUSY bit never was set if we get an OPERATION COMPLETE anyway.
4. Wait for the OPERATION COMPLETE code from the bubble memory controller. If this does not occur within a reasonable time, the command did not succeed.

The phrase "a reasonable time" in this subroutine means that the bubble memory controller's status was inspected BTRIES times without success. We have written the subroutines such that they will regard the command as having been successful if the bubble memory controller returns an OPERATION COMPLETE code even if the BUSY bit remains 0. ([Ref. 1] does not suggest that this latter indication can occur. However, if the command is accepted and completed very quickly, the control program might never observe the BBUSY bit, so it seems to be a good idea to permit it.)

It was our intention that this subroutine be used to issue *all* commands to the bubble memory. However, it executed too slowly to permit its use with data transfers. The subroutines `bubread()` and `bubwrite()`, written in assembly language, were

written for this purpose. In the case of other commands, TRUE is only returned if the bubble memory returns OPERATION COMPLETE in the status byte. FALSE is returned otherwise.

2. File `bubrw.s`

a. `char bubxfer(void)`

This subroutine is required during initialization of the bubble memory. It writes 40 0xff characters to the bubble memory. It returns TRUE if the transfer worked; FALSE otherwise. The subroutine is written in assembly language for speed, but is called in the same manner as a C subroutine.

b. `char bubread(char *buffer)`

This subroutine reads data from the bubble memory and places it in a buffer whose address is passed as a parameter. It is written in assembly language in order to execute sufficiently rapidly to preclude overflowing the buffer in the Bubble Memory Controller (BMC). The listing of this subroutine includes many comments which explain the purpose of each step. The following is a list of the actions which must be accomplished by this subroutine.

1. Save the contents of all registers.
2. Issue the FIFO reset command to the BMC.
3. Issue the READ command to the BMC.
4. Wait for the BUSY bit to become a one. If this never happens, the command has failed.
5. Input 64 characters from the bubble memory. The FIFO bit must be set to 1 before each character is read. If this bit never becomes a 1, the command has failed.
6. Restore the contents of all registers to what they were before the subroutine began to execute.

c. `char bubwrite(char *buffer)`

This subroutine writes data to the bubble memory from a buffer whose address is passed as a parameter. It is written in assembly language in order to execute sufficiently rapidly to preclude having the Bubble Memory Controller (BMC) empty its internal buffer before all the data has been sent to it by the experiment controller. The listing of this subroutine includes many comments which explain the purpose of each step. The following is a list of the actions which must be accomplished by this subroutine.

1. Save the contents of all registers.
2. Issue the FIFO reset command to the BMC.

3. Issue the WRITE command to the BMC.
4. Wait for the BUSY bit to become a one. If this never happens, the command has failed.
5. Output 64 characters to the bubble memory. The FIFO bit must be set to 1 before each character is written. If this bit never becomes a 1, the command has failed.
6. Restore the contents of all registers to what they were before the subroutine began to execute.

3. File clock.c

a. *void clockint(struct datetime *clock, struct idatetime *iclock)*

This subroutine takes a **datetime** structure pointed to by **clock** and converts it to an **idatetime** structure pointed to by **iclock**. The function **bcd_int()** is used to convert the binary coded decimal (BCD) format used in the **datetime** structure into the integer format used in the **idatetime** structure.

b. *char clockcompare(struct idatetime *clock1, struct idatetime *clock2)*

This subroutine compares the two times pointed to by **clock1** and **clock2**. It will return TRUE if the first time is equal to or later than the second; FALSE otherwise. To do the comparison, each element of the time is compared, from month down to second, in that order. The principle difficulty is in comparing dates that span New Year's Day. We want January 1 to be regarded as coming after December 31, not before.

To do this we first subtract the second month from the first. The difference is taken modulo 12. The modulo operation would not change any difference from 0 through 11; a difference of -11 through -1 would be changed to 1 through 11 respectively. Results in the range 1 through 5 indicate that the first date is later than the second.

For example, if the first date is January (month 1) and the second date is December (month 12), the difference is $1 - 12 = -11$. When this is taken modulo 12, we get 1. Thus January is 1 month after December.

If the first date is June (month 6) and the second date is December (month 12), the difference is $6 - 12 = -6$. Taken modulo 12, this is 6. Since this is greater than 5, we regard June as coming before December, not after.

c. *void clocksum(struct idatetime *result, struct idatetime *clock1, struct idatetime *clock2)*

This subroutine adds together the date and time pointed to by the **idatetime** structure **clock1** to the number of months, days, etc. in the **idatetime** structure pointed

to by **clock2**, yielding a new **idatetime** structure pointed to by **result**. This is useful when from a given date and time one wishes to calculate a later date and time. The usual use of this subroutine is, given the current date and time, to calculate the date and time after some given delay has elapsed.²⁵

It starts by adding the seconds together, and works from there up to the months. After each addition, checks are made to ensure that the result is valid. If not (e.g., 63 minutes is not valid), the result is corrected and any excess is carried over to the next highest unit of time.

The fact that different months have different lengths is a bit of a nuisance which is overcome by considering the three possible cases: a month can have 31 days, 30 days or 28 days. Leap years are ignored, since the real-time clock does not store the current year, and so is unaware of leap years.

d. *void show_waketime(struct idatetime *waketime)*

This function displays the date and time stored in the **idatetime** structure pointed to by **waketime** on the terminal.

e. *void dump_iclock(struct idatetime *clock)*

This subroutine displays the date and time (when stored in integer format) on the terminal.

f. *void get_time(struct idatetime *clock)*

This subroutine asks the user for the date and time. Each response is checked for correctness to preclude invalid dates and times being entered. The function **getint()** is used to get the responses from the keyboard. The responses are converted to binary-coded decimal (BCD) format and stored in the structure whose address is passed as a parameter to the function.

g. *void show_waketime(struct idatetime *waketime)*

This subroutine displays on the terminal the date and time when a time-out will have been completed. The date and time are provided in the structure whose address is passed as a parameter.

h. *char timeout(int delaytime, int measure)*

This subroutine has two purposes:

1. It initiates a time-out sequence.
2. It checks to see whether a time-out sequence has been completed yet. This is the case when the wake-up time calculated previously has been reached.

²⁵ While it could be used to add two *dates* together, this would not be particularly meaningful.

The subroutine calls the function `allow_ctrl_interrupts()` to permit its being interrupted during ground testing by the depression of a key on the terminal. It then calls `clockread()` to get the current date and time. This is converted to integer format by a call to `clockint()`. If the parameter `delaytime` is the constant `NULL`, then the function's purpose is to see whether a time-out set previously has expired. The function `clockcompare()` is invoked to compare the stored date and time with the current date and time. Its result is returned as the result of `timeout()`.

If the parameter `delaytime` is *not* the constant `NULL`, then the function's purpose is to initiate a time-out sequence. The structure `waittime` is initialized to zero. One of its elements is then modified to contain the number of delay units passed as the parameter `delaytime`. Which element is modified is determined by the parameter `measure`, which can take on the values `MONTH`, `DATE`, `HOURS`, `MINUTES`, or `SECONDS`. The subroutine `clocksum()` is called to add together the current time and the amount of time to wait. The result is displayed by calling the function `show_waketime()`. The wake-up time is stored in a global structure, `waketime`, so its contents will be undisturbed the next time this function is called.

4. File `convert.c`

a. `char atoh(char *ascii)`

This function converts the two-character hexadecimal string pointed to by `ascii` and converts it to a single character.²⁶ If the characters in `ascii` are in the range '0' through '9' or 'a' through 'f', then they will be properly interpreted as hexadecimal digits. Capital letters ('A' through 'F') and any other characters are treated as zeros. For example, the character string "63" would be converted to the single character 'c', since the hexadecimal representation of this ASCII character is 0x63.

b. `unsigned int atohexint(char ascii[ ])`

This subroutine converts a four-byte ASCII string of characters which represent a valid hexadecimal word into a single unsigned integer. No checks are made to see that the character string *is* valid, but invalid characters are sufficient to cause the subroutine to stop processing the character string. If no valid string of hexadecimal characters is found, the value 0 is returned by this subroutine.

c. `int atoi(char *s)`

This function converts a four-character string to an integer. The string may optionally include a sign (+ or -) in the first position. Successive characters will be

²⁶ No check is made to ensure that `ascii` is only two characters in length.

converted to numeric values if they are in the range '0' through '9'. Conversion ceases as soon as a character fails to fall within this range. No checking is done to ensure that the number of digits provided can fit within the number of bytes reserved for integers. This subroutine is from Bilofsky [Ref. 18].

*d. char *bcd_asc(char bcd)*

This function converts a binary coded decimal (BCD) character into an ASCII string. For example, the single byte 0x63 is converted to a two-character string "63". The BCD character is first converted to an integer. It is assumed that an integer occupies two bytes. If the leading nibble of the character is a 0, it will be converted to a space (' '). No check is made to see if the BCD character is valid. The function returns a pointer to the ASCII string representation.

Since it always uses the same storage location to hold the converted result, the string should be copied to another variable before `bcd_ascii()` is called again, for the old contents will be destroyed.

e. int bcd_int(char bcd)

This function converts a binary coded decimal (BCD) character into an integer. No checking is done to ensure the BCD character is valid. The integer result is returned.

*f. char *ctoh(char byte)*

This function converts a character byte into a hexadecimal string representation. It returns a pointer to the string. Since it always uses the same storage location to hold the converted result, the string should be copied to another variable before `ctoh()` is called again, for the old contents will be destroyed.

g. char int_bcd(int decimal)

This function converts an integer into BCD format. It will handle positive integers in the range 0 through 99. No checking is done to ensure the integer falls within this range. The function returns the BCD result as a single character.

*h. char *itoa(int n, char[])*

This function converts an integer `n` into an ASCII representation. It converts the integer into digits by taking it modulo 10 and storing the digits in character form in reverse order. Upon completion, it reverses the string in place. The result is stored in the location pointed to by the parameter `s`. No check is made to ensure this location has enough storage. This is the user's responsibility. However, in a machine with two-byte integers, the largest possible integer will contain five digits. The user

should allow two extra locations for the sign and the terminating NULL character, or seven characters in all. This subroutine is from Bilofsky [Ref. 18].

i. char tolower(int c)

This function converts upper case alphabetic characters into lower case ones. This subroutine is from Bilofsky [Ref. 18]. Its use here is a consequence of our having used this C compiler during the early work on this project. This function is provided in the library supplied with the Uniware C Compiler; with the Toolworks C Compiler, its source code had to be included with our source code. It could have been removed when we switched over to using the Uniware C Compiler, but it never was. There exist other vestiges of our early use of the Toolworks C Compiler that have never been eradicated.

*j. char *uitoh(unsigned int word)*

This subroutine converts an unsigned integer to hexadecimal ASCII string format. For example, the unsigned integer '28' is converted to the character string "1C" since that is its ASCII representation.

5. File delay.s

a. void delay(int n)

This function provides a timing delay of $n \times 10$ ms . It is written in Z-80 assembly language. It will only work correctly if the system clock has frequency $f = 2.5$ MHz. It is adapted from a program written by Mr. David Rigmaiden of the Naval Postgraduate School.

6. File expmnt.c

a. char ad_read(char port)

This subroutine will obtain one character from that channel of the Analog-to-digital (A/D) Converter whose address is pass as the parameter **port**. It functions by writing anything at all to that port address (we chose arbitrarily to send a 0), then by waiting for one 10 ms period during which the A/D converter responds, and then finally by reading a character from the same port. This is the character returned by the subroutine.

b. int adtoint(char addata, unsigned long multiplier)

The purpose of this subroutine is to convert a character input from the Analog-to-digital (A/D) Converter into an integer which represents the measured quantity in more meaningful units than an arbitrary number in the range [0,255], which is all that the A/D converter can provide. It would be natural to perform the arithmetic scaling of the input eight-bit number **addata** to the corresponding output value by per-

forming ordinary floating-point multiplication and division. This has one drawback in a microprocessor application: the executable code to support floating point operations takes up rather a large amount of memory. In tests we performed using the floating point arithmetic operators provided with the Uniware C Compiler, the subroutines required two extra EPROMs of 8 KBytes apiece. We had the room in our controller to accept this, but chose not to do so since our need for floating point arithmetic was limited to this one function. To program two complete EPROMs every time a new version of the program was compiled solely to provide this one use of floating point of arithmetic was not warranted, in our judgement.

The alternative was to perform the scaling operation with fixed point arithmetic. The C programming language supports integer operations, but fixed point operations with a movable decimal point are not supported. This subroutine uses an implied decimal point. As far as the subroutine is concerned, the operands are integers, plain and simple. By using unsigned long integers, we have 32 bits of accuracy, permitting numbers in the range $[0, 4.294967296 \times 10^9]$. For our purposes, we have used a divisor of 10^6 . This has the effect of reducing the range of useful numbers to $[0, 4294]$.

The purpose of these manipulations is to permit all the accuracy promised by the provision of eight bits from the A/D converter while avoiding floating point arithmetic. The details of the operation are included as comments in the program listing, so will not be repeated here.

c. *void alter_page0(struct page0data * pagezero)*

This subroutine permits the user to alter the flags stored in page 0 of the bubble memory. Since these flags describe the current status of the experiment, and also indicate which area of the bubble memory is available for use, their alteration amounts to initializing the status of the experiment to a known value. The use of this subroutine is described in Chapter V. HOW TO GET THE EXPERIMENT READY FOR A LAUNCH on page 63. It displays a menu using the same method used in so many other functions in the control program. This method was already presented in the description of the function `main()`. A `while` loop presents the menu repeatedly until choice Z is made. The menu shows the current values of all the information stored in page 0 of the bubble memory. In most cases it also shows the other possible value of each flag. The sole exceptions are the value of the next available page, which can fall in the range $[1, 8191]$, and the flag `RECORD_start_time`, which is a date and time.

If any of the values is changed, page 0 is rewritten.

d. *char bad_idea_to_record(char show)*

In the abridged experiment, the only phase of the experiment is *record*. It would be unfortunate if the Solid State Data Recorder were restarted in *record* mode *after* the launch had already occurred, for this would erase the recording of the launch and replace it with the silence that fills the cargo bay in space, or possibly all the sounds of the launch except the moment of ignition of the solid rocket motors. How could this happen? A power fault, for whatever reason, would cause the controller to start at the beginning upon the restoration of power. There is no obvious way for the controller to be sure that it is still on the ground, which certainly is the only time when it is really a good idea to initiate the *record* phase. Of course, it is easy to determine whether it is in space or not simply by checking the status of the barometric pressure switches. Our solution is simply to prohibit the initiation of *record* mode more than once in 12 hours. If a mission is scrubbed, it will be at least 24 hours before it is rescheduled. The 12 hour delay will have elapsed by this time, and the abridged experiment could then be performed as planned.

The long and the short of these considerations is that this function compares the current time to the time when *record* was last initiated. This time is stored in page 0 of the bubble memory. If insufficient time has elapsed, the function returns the value TRUE, meaning that it *is* a bad idea to record. If 12 hours has elapsed, it returns the value FALSE, meaning it is *not* a bad idea to record; *record* mode can be initiated, in this case.

e. *void display_page0(struct page0data * pagezero)*

This subroutine displays the contents of page 0 of the bubble memory on the terminal.

f. *void do_sweep(void)*

This function performs the *sweep* phase of the unabridged experiment. It turns on (and logs the fact that it has done so) the Analog-to-digital (A/D) Converter and the Solid State Data Recorder (SSDR). It then commands the SSDR to enter *sweep* mode. After a 10 second time-out, it applies power to the Voltage Controlled Oscillator (VCO) which is responsible for filling the Space Shuttle cargo bay with sounds of known frequency.

Next it initiates a 13 minute time-out. The SSDR should signal completion of the *sweep* phase before this much time has elapsed. If this does not happen, the time-out allows the control program to stop waiting for it to do so. Upon completion

of the *sweep* phase, the `do_sweep()` phase removes power from the VCO, SSDR, and A D converters.

g. char initialize(void)

This subroutine extracts the status information from page 0 of the bubble memory when the `expmnt()` program begins to execute. It will remove power from the Voltage Controlled Oscillator (VCO), which performs the *sweep* phase, and from the heater subsystem, if either of these is on. It will not remove power from the other subsystems, which also might be on when power is first applied. How can this be, and why does it not remove power from them? One way in which power might be applied is after a brief power fault. If the fault affected the controller but not, say, the Solid State Data Recorder (SSDR), and if *sweep* mode had been initiated prior to the loss of power, removing power from the SSDR would have the effect of terminating *sweep* mode and this would raise the possibility that an otherwise successful recording of the ignition of the solid rocket motors would be foiled. The SSDR and other subsystems, therefore, should not be interfered with at this point.

h. char listen(void)

This subroutine applies power to the matched filter circuit board. It then calls `we_launched()`. This function returns DLAUNCH if a launch has occurred, and this value is returned by `listen()`, too. If a launch has not yet occurred, `listen()` checks to see if the matched filter has detected the starting of the Auxiliary Power Units (APUs). If so, the function returns the value DAPUON. If neither condition has occurred, the function calls the subroutine `look_ahead_discard()`, giving the user (if any) to get out of the `listen()` function by depressing any key on the terminal. Barring one of these three conditions, the function will continue making these same checks indefinitely.

i. char logevent(char event)

This subroutine makes coded entries in the bubble memory of all events which take place. While it is doing this, it takes readings from each of the channels of the Analog-to-digital (A D) Converter and stores the results in the same page of the bubble memory in which the event code is stored. If the bubble memory is already full, which would occur after $2 \times 8191 = 16,382$ events, the subroutine refuses to store any more events. This will preclude the destruction of the records of earlier events. However, we do not expect this many events ever to occur on a single mission of the Space Shuttle. The interval between successive events after the first two minutes of flight is five minutes; at this rate, it would take nearly 57 days to fill up the memory.

There are two possibilities when the function prepares to write a page of information to the bubble memory. Either this is a brand new page, or it is the second half on an existing page of stored data. In the former case, the second half of the page is filled with NULL characters. This effectively erases any data previously stored in this upper half-page. The subroutine next reads the current date and time and stores this in the structure where all the information is assembled prior to being transferred to the bubble memory. The event code is passed to the function `logevent()` as parameter `event` to be stored in the bubble memory. Each channel of the A/D converter is sampled and the results also are stored in the bubble memory. If the event code is CSSWEEP, then the command to initiate *sweep* mode has just been issued, and the flag `sweepstarted` in page 0 needs to be set to TRUE. If the event code is DPRESSURE, then the flag `launchdone` in page 0 needs to be set to TRUE. Then the new record of information can be written to the next available half of the next available full page of the bubble memory. The page number and half page number are extracted from page 0 of the bubble memory. Page 0 needs to be updated, and this is done also.

j. `void log_menu(void)`

This subroutine provides the user with a menu for changing the contents of page 0 of the bubble memory. Recall that this information describes the current status of the experiment. It is important that this information be initialized correctly prior to the launch of the Space Shuttle. How to do this is described in Chapter V. HOW TO GET THE EXPERIMENT READY FOR A LAUNCH on page 63.

The details of how the menu is generated are the same as explained in the description of the function `menu()` and will not be repeated here.

k. `void monitor_heaters(void)`

This subroutine has the job of maintaining the temperature of the bubble memory at a sufficiently high level that it can be operated safely. If it finds that the current temperature is lower than the minimum desirable temperature (12°C) it will turn the heaters on. If it finds that the temperature is above the maximum desirable temperature (14°C), it will turn the heaters off. This is not the temperature above which the bubble memories will lose their memory. Rather it is a temperature chosen to be slightly higher than the minimum desirable temperature. If the temperature can attain this level, the heaters will be shut off for a while to save power. If the temperature falls to the minimum desirable level, this still is 2°C above the minimum operating temperature, allowing a reasonable margin for safe operation. The 2°C spread is wide enough to preclude excessively frequent operation of the relay switches, too.

l. void post_launch(void)

This subroutine performs the caretaking functions that follow the successful launch of the Space Shuttle. Its first action is to remove power from all subsystems. It then initiates a five minute time-out. During this wait, it calls `monitor_heaters()` repeatedly to give them an operation to operate the heater subsystem. It also checks the barometric pressure switches if they have not yet reported a completed launch. During ground testing it is useful to have a way of interrupting this phase of the mission. Calling `look_ahead_discard()` lets the user do so by pressing any key on the terminal. At the completion of the five minute delay, `logevent()` is called to record a "read A.D" event. The function `logevent()` takes care of reading all the Analog-to-digital Converter (A D) channels whenever it is called. Finally, a call to `voltages_low()` is made to ensure that if the voltages on the 10V power busses falls to too low a level, the experiment will be terminated. This will preclude an attempt to operate the bubble memories with insufficient current, which could cause them to lose their contents.

m. void record(void)

This subroutine performs the *record* phase of the abridged experiment. The first action taken by this subroutine is to read the current date and time and to place this information in the structure of data to be stored in page 0 of the bubble memory. A call to `logevent()` immediately after this has the effect of ensuring that this date and time are transferred to page 0 right away, along with taking current readings of all the channels of the Analog-to-digital (A D) Converter.

The `record()` subroutine then applies power to the A D converters and the Solid State Data Recorder (SSDR), commands the SSDR to enter *record* mode, and initiates a 20 minute time-out. The SSDR should report completion of *record* mode prior to the expiration of this delay, but even if it fails to do so, the subroutine will be able to terminate the *record* phase of the experiment. While waiting for the 20 minutes to elapse, the subroutine calls `baro_switch()` if that function has not previously reported a successful launch. Upon the completion of the *record* phase, the subroutine removes power from both the SSDR and the A D converter.

n. void short_experiment(void)

This subroutine performs the abridged experiment. It first checks to see whether a launch had occurred previously. This could be the case if a power fault had caused the controller to start executing its program from the beginning. If a launch has been recorded already, the subroutine refuses to put the Solid State Data Recorder

(SSDR) into *record* mode. This will prevent the successful recording of a launch to be wiped out.

If a launch has not occurred previously, then the subroutine will wait until it is alright to initiate *record* mode. This is indicated by the subroutine **bad_idea_to_record()** returning the value FALSE, meaning it is not a bad idea to start *record* phase. Next the subroutine will call **listen()** to listen for the starting of the Auxiliary Power Units (APUs). The **listen()** subroutine will keep control until either the APUs start, or until some indication of a launch is detected. At this point, the *record* phase is initiated.

It is conceivable that at the end of the *record* phase, we would discover that we had jumped the gun and that the Space Shuttle was still on the ground. To see whether or not this is the case, the subroutine calls **baro_switch()**, if that subroutine had not previously reported that launch had definitely been completed. If no launch has occurred, we are in a bit of a quandary. Is launch imminent? How long will it be before it occurs? It would be nice simply to re-initiate *record* mode, but this consumes considerable power. What is potentially worse, the power fault might occur at the moment of launch. It will still be several moments before the barometric switches indicate that a launch has occurred. Since we cannot ascertain whether the launch has occurred or not, it is best to assume that it has and not to re-initiate *record* phase, which would erase the recording of the launch. So we have adopted the solution of waiting until at least 12 hours more have elapsed before entering the *record* phase again.

At the successful completion both of *record* phase and a launch, the **short_experiment()** subroutine calls **post_launch()** to perform all the caretaking functions required during the Space Shuttle's mission.

o. void show_event(char event)

This function is used to display event codes stored in the bubble memory log in a readable form on the display terminal. It does this by displaying the appropriate character string from an array of strings which describe the various codes.

p. void shut_down(void)

This function removes power from any subsystem which currently is receiving power. It calls **logevent()** to record any actions it takes.

q. char ssdrmode(char mode)

This subroutine issues commands to the Solid State Data Recorder (SSDR). If the command is unsuccessful the first time, it will make several more tries before giving up. Once the command has been issued, the subroutine waits for 20 ms and then it

checks the status code returned by the function `ssdr_status()`. The desired response from the SSDR is that the commanded operation has been completed successfully, which is indicated by the return of the constant `NORMOP`. The subroutine `ssdrmode()` returns `TRUE` if this occurs; `FALSE` otherwise.

r. char ssdr_status(void)

This subroutine reads the status code from the Solid State Data Recorder (SSDR) and returns it to the calling function.

s. char voltages_low(void)

This function checks the channel of the Analog-to-digital (A/D) Converter which allow the measurement of voltage on the 10V bus. The value read is converted to voltage by the function `adtoint()`. If that voltage falls below the minimum voltage desirable on the 10V bus, then the function returns the value `TRUE`, meaning that the voltages are too low, and that the experiment should halt. Otherwise it returns the value `FALSE`.

t. char we_launched(void)

This subroutine first calls the function `baro_switch()` to see whether the barometric pressure switches have detected an ascent of the Space Shuttle. If this has occurred, or if the Vibration-activated Launch Detector has detected a launch, this function returns the value `DLAUNCH`. Otherwise it returns the value `FALSE`.

7. File `fputc.c`

*a. int fputc(int chr, void *device)*

The UNIWARE compiler provides the standard C output subroutine `printf()` to provide output to the standard output device. However, this subroutine requires the user to provide a subroutine `fputc()` to handle the output of a single character to any arbitrary device. We only support output by `fputc()` to the RS-232C terminal, so this subroutine is specific to that device.

This function calls the subroutine `allow_ctrl_interrupts()` to permit the user to interrupt operation of the control program. The subroutine will *not* output a character if, upon checking, it finds there is no terminal attached to the serial interface port. Thus, when the experiment is operating, calls to `printf()` are of no effect unless there is a terminal connected.

The subroutine returns `-1` if there is no terminal connected. This is the code specified by UNIWARE if `fputc()` is unable to do the output operation. If there is a terminal attached, `fputc()` repeatedly polls the serial interface, waiting for it to be

ready to accept output data. It then outputs the character, and returns that character, again as specified by UNIWARE. [Ref. 17: Compiler Section, pp. 45 and 52]

8. File `global.c`

This file contains the declarations of variables used throughout the control program. The author's predilection is to avoid the use of global variables. However, it can sometimes become so awkward to observe this preference as to make it silly. It is desirable to hold the use of global variables to a minimum, however.

9. File `inout.c`

a. *`void allow_ctrl_interrupts(void)`*

This subroutine makes it possible for the user to interrupt the execution of the control program. Whenever it gets control, it calls the function `look_ahead()` to see if any key of the terminal has been depressed by the user. If not, then the function returns without further ado. If a key *has* been depressed, however, it may have been one of the two control keys CTRL Y or CTRL S. If so, the function `termin()` is called to remove the character from the input buffer, and to respond appropriately to the input control character.

b. *`void dump(unsigned int address, unsigned int length)`*

This subroutine displays the contents of a section of memory on the display terminal. The variable `address` designates the address of the first character of data to be displayed. The variable `length` specifies how many characters to display. The display shows a hexadecimal representation of every character in the chosen section of memory, and if that character has a printable form, that form also is displayed. This function is of value only for debugging the control program.

c. *`char gethex(void)`*

This subroutine obtains a hexadecimal string from the terminal. Up to HSTRLEN characters will be accepted. Processing will cease as soon as a character not in the ranges '0' through '9', 'a' through 'f', or 'A' through 'F' is entered. The input string will be converted to a single character by calling `atoh()`, and this character will be returned. For example, the string "6a" would be converted to the ASCII character 'j', whose hexadecimal representation is 0x6a. This subroutine is useful for getting one-byte system port addresses from the user if he is more likely to know them in hexadecimal than in decimal.

d. *`unsigned int gethexint(void)`*

This subroutine is very similar to `gethex()`, except that it accepts two hexadecimal bytes, not just one.

e. int getint(void)

This subroutine obtains a decimal string from the keyboard. Up to STRLEN - 1 digits can be entered. Processing will cease as soon as a character not in the range '0' through '9' is entered. The input string will be converted to an integer by calling `atoi()`, and this value will be returned.

f. int getpageno(void)

This subroutine obtains a page number in bubble memory from the user. Valid responses are in the range 0 through MAXPAGE. It uses the subroutine `getint()` to obtain the response.

*g. char look_ahead(char *character)*

This function checks to see if a key has been pressed on the display terminal. Of course, if there is no terminal attached, there is no point in even looking, so the function returns instantly in this case with a value of FALSE. The variable `console_data_available` is one of two variables known to all functions in the file `inout.c`. It will have the value TRUE if the function `look_ahead()` or the function `termin()` discovered previously that there was a character available to be read. The `look_ahead()` returns this character to the calling function for it to inspect, but it does not remove the character from the buffer. Further calls to `look_ahead()` or to `termin()` would obtain the same character.

If there is no character already in the buffer (that is, if `console_data_available` is FALSE,) then `look_ahead()` checks the RS232C interface to see if a key has been pressed. If so, the character is read and placed in the variable `console_buffer` for future use by `look_ahead()` and `termin()`. It also is returned to the calling function, and the value of `console_data_available` is set to TRUE since a character now is in the console buffer.

h. char termin(void)

The primary purpose of this subroutine is to read a character from the terminal whenever the latter has one available. This condition is known to be true whenever bit PRTRDY of port PRCTRL is a 1. The input character is returned to the calling function.

In order to permit the control program to be interrupted, however, the function `termin()` interprets the characters CTRL S and CTRL Y specially. CTRL S is interpreted to mean "stop displaying data on the display terminal" if data is being displayed, or "start displaying data on the display terminal" if the display has already been halted by CTRL S. In other words, the CTRL S switch operates as a toggle switch to

stop and start the display of data. CTRL Y is interpreted to mean “call the diagnostic subsystem menu”. We do not wish this to be done more than once at a time, for otherwise we might make so many recursive calls to the program `menu()` that the stack would be corrupted.

The variable `allow_menu_call` will be TRUE the first time `termin()` is called. If CTRL Y is entered, `allow_menu_call` is set to FALSE, and further calls to `menu()` are precluded thereafter. It is only returned to the value TRUE if the `menu()` program is completed by the user later.

The variable `waiting_for_ctls` will switch from FALSE to TRUE or back again each time CTRL S is entered by the user. Data from the keyboard will only be accepted when this variable is FALSE, in which case the display has not been halted.

The variable `ctrl_valid_data` will be a copy of the variable `console_data_available` described earlier.

If no data has been read into the console buffer previously by `termin()` or by `look_ahead()`, then `termin()` will wait until a character is available. Once this occurs, the variable `console_data_available` is set to FALSE, since `termin()` has filled and emptied the console buffer all at once. A switch statement allows the character to be interpreted.

i. void testinput(void)

This subroutine asks the user to specify a port address in hexadecimal. It then reads a character from that port and displays it on the terminal.

j. void testoutput(void)

This subroutine asks the user to specify a port address in hexadecimal, and then asks for a hexadecimal byte to be sent to that port. The data is accordingly output to the port.

10. File `main.c`

a. void memory_dump(void)

This subroutine asks the user for the first address in memory whose contents he wishes to inspect, and for the number of characters which he wishes to see displayed. It then calls the subroutine `dump()` to honor the request. This function is only useful for very low-level debugging of the software.

b. void testio(void)

This subroutine presents a menu permitting the user to send data to any port, and to read data from any port, in the system. The method of implementing a menu is the same as that presented in the description of the function `menu()` and will not

be repeated here. For output to a port, the function `testinput()` is called. For input from a port, the function `testoutput()` is called.

11. File `mbrk.s`

a. *char *mbrk(long size, long *realsize)*

This subroutine is written in Z-80 Assembly language. It is described in [Ref. 17: Compiler section, p. 51], from which it is drawn.

12. File `newio.s`

a. *char input(char port)*

This subroutine is written in Z-80 Assembly language. It inputs a character from a port and returns it to the calling function.

b. *void output(char port, char data)*

This subroutine is written in Z-80 Assembly language. It outputs a character to a port.

13. File `power.c`

a. *void power_status(void)*

This subroutine returns the status byte from the POWERIN port. This status is described in Table 3 on page 16.

b. *char power_write(char command)*

This subroutine issues commands to the power circuit board. Valid commands are SSDROFF and SSDRON (to turn the Solid State Data Recorder off and on); VCOFF and VCON (to turn the Voltage Controlled Oscillator off and on); ADOFF and ADON (to turn the Analog to Digital Converter board off and on); MATOFF and MATFON (to turn power to the matched filter, launch detector and barometric switch off and on); and HEATOFF and HEATON (to turn power to the heater circuit off and on).

A command can be executed by writing it to the POWEROUT port and then setting bit PWRSTROBE in port C_1 to a 1. A delay of length PWRDELAY is required before bringing that bit to 0 again. Another delay of the same length is then required. These delays ensure proper functioning of the relays. Each bit in the status byte returned by the function `power_status()` indicates whether the associated relay is on or off. The bit is 0 if the relay is on; 1 otherwise. The `power_write()` function examines the bit corresponding to the relay it attempted to switch. A TRUE is returned if the relay is in the desired position; FALSE is returned otherwise.

14. File start.s

This file contains the controller's start-up code. It is written in Z-80 Assembly language. Whenever the Z-80 receives power, it starts executing from location 0x0000. The start-up code initiates the stack pointer to the value STACKTOP and then causes a jump to START. All other Z-80 interrupt locations are initialized such that they cause a jump to the same location, since interrupts are not used by the controller. At START, the ix register is initialized to 0. This register is used by the C compiler to point to parameters and local variables within C programs.

Memory may be requested by C programs using the `mbrk()` function provided with the UNIWARE C Compiler. The start-up code uses a variable MBRKPTR to point to the next available address of allocable memory. Initially this variable is set to MRAM, a global variable set in the file `\vibro\control\object\spec` to point to the beginning of all allocable memory. Once `mbrk()` has obtained some memory, it keeps it, so the start-up code never needs to reclaim it. Consequently, MBRKPTR can only increase; it can never decrease.

ZRAMSZ is the number of RAM locations starting at ZRAM which are used for uninitialized, static variables in the C programming language subroutines. The start-up code writes zeros to all these locations, because the C programming language specification is that uninitialized static and external variables be initialized to 0 by the compiler [Ref. 16: p. 198].

IRAMSZ is the number of RAM locations starting at IRAM which will contain initialized data. This data is stored in ROM locations starting at RAMDATA at the time the program is burned into ROM. The start-up code copies it from ROM to RAM. Finally, control is passed to `main()`, the user's C program. If `main()` should ever return control to the start-up code, a halt instruction is executed. The start-up code is adapted from an example given in [Ref. 17: Compiler Section, pp. 13-15.]

Table 12. CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\BATCH

File	Contents
ASM.BAT	This batch file simplifies the assembly of Z-80 assembly language source code. To use it, type <code>asm <source filename> .s</code> . For example, to assemble the file <code>delay.s</code> , type <code>asm delay.s</code> . Note that the file type need not be <code>s</code> , but whatever it is, it must be present. Use of <code>s</code> is recommended for clarity. The procedure produces object files in subdirectory \vibro\contrlr\object and assembly code list files in subdirectory \vibro\contrlr\list. See the description of the batch file <code>asmlist.bat</code> for instructions on how to produce this listing file, which includes all addresses supplied by the linker.
ASMLIST.BAT	This batch file produces a listing of the assembly language source file generated by the Z-80 assembler. To use it, type <code>asmlist <filename.filetype></code> . The output is appended to the file \temp\print. It can be printed by use of the batch file <code>promout.bat</code> . These listings include all global addresses supplied by the linker, provided \vibro\contrlr\vibro.out has been generated by <code>promlink.bat</code> .
C.BAT	This batch file simplifies the compilation of C source code. To use it, type <code>c <source file name> .c</code>. For example, to compile the file <code>vibro.c</code>, type <code>c vibro.c</code>. Note that the file type need not be <code>c</code>, but whatever it is, it must be present. Use of <code>c</code> is recommended for clarity. The procedure produces object files in subdirectory \vibro\contrlr\object and assembly code list files in subdirectory \vibro\contrlr\list. See the description of the batch file <code>asmlist.bat</code> for instructions on how to print listing files, which show all addresses supplied by the linker.
LIST.BAT	This batch file produces a listing of any MS DOS file. To use it, type <code>list <filename.filetype></code> . The output is appended to the file \temp\print. It can be printed by use of the batch file <code>promout.bat</code> .

LOADMAP.BAT	This batch file appends a copy of the load map into <code>\temp\print</code>. It can be printed by use of the batch file <code>promout.bat</code>. The load map shows the absolute addresses at which the eight regions of code produced by the compilation, assembly, and linking steps are placed. It is useful to have this so that you know whether the controller has enough RAM and ROM installed to hold the output program. The listing shows the starting address of each region and the number of bytes it occupies. Regions <code>reset</code>, <code>code</code>, <code>const</code>, <code>string</code>, and <code>data</code> all must be stored in ROM initially. Of these, only <code>data</code> belongs in RAM eventually, yet it must be stored in ROM initially. The reason is that it contains C variables whose values have been initialized. If they were not stored in ROM, those values would not be available at execution time. The start up routines in <code>\vibro\contrlr\asmsource\start.s</code> cause these initialized variables to be copied from ROM to their proper locations in RAM. These locations are those shown in the load map. Thus, in addition to the ROM space required for the other four regions, be sure to allow enough room for the data region to be loaded into ROM, too. For example, if there is only one 8K ROM installed at location 0x0000, but the load map shows that more than 8K of ROM is required, then there is insufficient ROM in place. Either more must be added, or the program must be reduced in size. How to load the executable program into ROM is described below in 2. Getting the Executable Program into EPROM on page 146.
PRINTALL.BAT	This batch file will produce a listing of all source files, all batch files, a load map, and a symbol listing. The output will be appended to the file <code>\temp\print</code>. Normally you would first empty this file using <code>readyout.bat</code>. After producing a complete listing, it could be printed on the printer using <code>promout.bat</code>.
PROMLINK.BAT	This batch file simplifies the conversion of the object modules into an executable output program. To use it, just type <code>promlink</code>. It creates two output files. The first of these is <code>\vibro\contrlr\vibro.out</code>. It contains information about the addresses assigned by the linker to global variables. This file is used by the batch file <code>promsym.bat</code>. The other file which <code>promlink.bat</code> produces is <code>vibro.hex</code> which can be loaded into an EPROM.
PROMOUT.BAT	This batch file causes the file <code>\temp\print</code> to be printed. The latter file contains the output of the list, <code>asmlist</code>, <code>loadmap</code>, or <code>promsym</code> batch file executions. It does not erase <code>\temp\print</code>. Use <code>readyout.bat</code> to do this.

PROMSYM.BAT	This batch file appends a listing of all the variables known globally throughout the control program to the file <code>\temp\print</code> . These include both C language source code variables, Z-80 assembly language global symbols, and several symbols defined by the linker specification file. This listing is useful in determining how variables have been declared and in finding the absolute addresses of symbols. It can be printed by use of the batch file <code>promout.bat</code> .
READY-OUT.BAT	This batch file should be used before using any of the following: 1. <code>list.bat</code> 2. <code>asmlist.bat</code> 3. <code>printall.bat</code> 4. <code>promsym.bat</code> 5. <code>loadmap.bat</code> Its purpose is to empty the temporary files <code>\temp\temp</code> and <code>\temp\print</code> prior to their being used by those other batch files. Once used, you need not use it again unless you have already printed the contents of the temporary file and need it no longer, or unless you wish to discard it for some other reason.

C. PROGRAM MAINTENANCE

This section describes how to compile a new version of the controller program; and how to get an executable version of that program into an EPROM. A basic familiarity with Microsoft MS DOS is assumed. The file organization is described in APPENDIX D. HIERARCHICAL ORGANIZATION OF SOFTWARE FILES on page 88.

1. Procedures for Generating a New Executable Program

a. *Compile the C source files*

For each source code file written in the C language, type `c <filename> .c`.

b. *Assemble the Assembly Code Source Files*

For each source code file written in Z-80 assembly language, type `asm <filename> .s`.

c. *Link Modules Together*

Enter the command `promlink`. This links all executable modules together, generating an executable program module in file `vibro.bin` in subdirectory `\vibro\contrlr`, which becomes the current directory upon completion.

Table 13. CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\CSOURCE

File	Contents
BUBBLE.C	Contains programs which operate the bubble memory module on the controller board.
CLOCK.C	Contains programs which operate the real time clock on the controller board.
CONVERT.C	Contains programs which perform conversion of data from one format to another.
EXPMNT.C	Contains programs which are specially designed for use with the Vibro-acoustic Experiment. They are not usable by other applications, although they might be tailored to them.
FPUTC.C	Contains the routine <code>fputc()</code> .
GLOBAL.C	Contains the declarations of the few variables which are declared with global scope (<i>i.e.</i> , which are known to all subroutines).
INITIAL.C	Contains programs which initialize both NSC810A RAM-I O-Timer chips on the controller board.
INOUT.C	Contains programs which handle input from and output to any device.
MAIN.C	Contains the highest level of programs which operate the controller, including the C subroutine <code>main()</code> . These include most of the menu-driven routines which are executed if there is a terminal attached to the controller when it receives power.
POWER.C	Contains programs which operate the electrical power relay board in the controller. This board supplies power to various hardware subsystems.

2. Getting the Executable Program into EPROM

a. Copy the Executable Program to a Diskette

Place a 5 1/4 inch diskette in drive B. Then enter the command `copy vibro.hex b:`. This puts a copy of the file `vibro.hex` on the diskette. This file contains a hexadecimal format of the code which, when loaded into an EPROM, will allow the controller to function.

b. Prepare to Write EPROMs

We have acquired the Intel program PCPP PC Personal Programmer to load data into EPROMs. Take the diskette to the IBM Personal Computer (PC) with the EPROM programmer. This PC is located in Space Lab #2, Room 102, Bullard Hall, Naval Postgraduate School. Be sure you have enough EPROMs available.

Table 14. CONTENTS OF SUBDIRECTORY
\VIBRO\CONTRLR\ASMSOURC

File	Contents
BUBRW.S	This file contains the routines bubread() , bubwrite() and bubxfer() . These routines had to be written in assembly language because the compiled code would not execute fast enough correctly to perform data transfers with the bubble memory controller. Each routine looks just like a C language subroutine to the calling routine.
DELAY.S	This file contains a delay routine written in Z-80 assembly code, but it can be called as if it were a C language subroutine. Its purpose is to provide delays in multiples of 10 ms in situations where the hardware requires it.
MBRK.S	This is a routine supplied with the UNIWARE C compiler. Its purpose is to allow C programs to request memory through the standard allocation routines malloc() and calloc() [Ref. 17: Compiler section, pp. 50-51].
NEWIO.S	This file contains the two routines input() and output() . They are written in Z-80 assembly code, but they can be called as if they were C language subroutines. They provide the ability to read characters from and write characters to any valid port address.
START.S	This file contains the Z-80 initialization code, such as an address where execution should begin, interrupt vectors, code for initializing RAM, and a call to the main() program, located in the C source file vibro.c . It is adapted from code provided by UNIWARE.

To ensure they are empty, place them in the EPROM eraser and turn on the fluorescent light to erase their contents. While this is going on, and once the PC is booted up, enter the command **cd pcpp** at the command line. This will make **pcpp** the current subdirectory, and so the program **pcplod** can be issued to initialize the program which will write the file **vibro.hex** into the EPROMs. Once this has been done, enter the command **ipps channel(3)**, which actually invokes PCPP.

PCPP now has control. Enter the following commands:

- t 2764** This command allows 2764 EPROMs to be used.
- i 80** This specifies that INTEL 8080 hex format files are being used. This is the format of the program in the file **vibro.hex**.

Table 15. CONTENTS OF SUBDIRECTORY \VIBRO\CONTRLR\HEADERS

File	Contents
BUBBLE.H	This file contains the extern declarations of the routines in bubble.c .
BUBRW.H	This file contains the extern declarations of the routines in bubrw.s .
CLOCK.H	This file contains the extern declarations of the routines in clock.c .
CONVERT.H	This file contains the extern declarations of the routines in convert.c .
DELAY.H	This file contains the extern declarations of the routines in delay.s .
EXPMNT.H	This file contains the extern declarations of the routines in expmnt.c .
GLOBAL.H	This file contains the extern declarations of the variables in global.c .
INITIAL.H	This file contains the extern declarations of the routine in initial.c .
INOUT.H	This file contains the extern declarations of the routines in inout.c .
MAIN.H	This file contains the extern declarations of the routines in main.c .
NEWIO.H	This file contains the extern declarations of the routines in newio.s .
POWER.H	This file contains the extern declarations of the routines in power.c .
VIBRO.H	This file contains definitions of all constants used by the C routines. It also contains definitions of global structures used throughout.

- b This performs a check to ensure the EPROM currently loaded in the socket is blank. It should be obvious that a blank EPROM must be inserted in the slot before performing this check.

c:vibro.hex (0000,1fff) t p

c:vibro.hex (2000,3fff) t p

c:vibro.hex (4000,5fff) t p

c:vibro.hex (6000,6c23) t p

c:vibro.hex (e000,e127) t p (0c24)

The last five commands copy the program instructions from the diskette into the EPROM. The numbers in parentheses are the addresses which are to be loaded into each EPROM. A new EPROM should be inserted into the socket prior to executing each of the first four commands. The number 0x6c23 in the fourth command is one less than the number RAMDATA in the symbol table. The number 0e127 is one less than the value of ZRAM in the symbol table. The number 0x0c24 is 0x6000 less than the number RAMDATA in the symbol table. This number tells the PCPP program where in the final EPROM to begin writing this section of data. Since the EPROM addresses all are in the range [0x0000,0x1fff], subtracting 0x6000 from the actual starting address is necessary to get the address into the proper range. Note that the last command causes the data which eventually will be placed in RAM to be loaded at the end of all the data which is to remain in EPROM locations. It is conceivable that this information would not fit onto the end of a single EPROM but might spill across the end and require another EPROM. This would require modifying the instruction sequence shown above. For details, consult [Ref. 20]. The command `exit` will terminate the operation of the PCPP program.

This completes the loading of the control program into EPROM. The EPROMs can now be loaded into the controller for testing.

APPENDIX H. CONTROL PROGRAM SOURCE CODE

A. FILENAME SPEC

```
/******
Specification file for the Controller hardware. Also see
companion files "start.asm" and "mbrk.asm". This specification
assumes 32K of ROM at address 0x0000, and 8K of RAM at address 0xe000.
******/
partition {
    overlay {
        region {} reset [addr = 0]; /* reset vector */
        region {} code, const, string; /* other ROM */
        RAMDATA = $; /* ROM to initialize region ram */
        /******
        Region ram is initialized at runtime startup by
        copying data from RAMDATA to IRAM. The data must
        actually be linked to its RAM address (IRAM) to get
        correct variable addresses, but must be
        programmed into ROM here. By hand, you must
        ensure that ENDDATA <= ENDROM. (ENDDATA is below)
        *****/
        ENDROM = 0x8000; /* end of ROM */

        IRAM = 0xe000; /* RAM starts here. */
        region {} data [addr=0xe000]; /* RAM to be initialized on reset */
        IRAMSZ = $ - IRAM; /* # bytes to copy from RAMDATA */

        ENDDATA = RAMDATA + ($ - IRAM); /* compare this against ENDROM */

        ZRAM = $; /* Pointer to start of ram region. */
        region {} ram; /* RAM to be zeroed on reset */
        ZRAMSZ = $ - ZRAM; /* # bytes to zero on reset */

        MRAM = $;
        region {} mbrkram[size=0x250]; /* RAM available to malloc() */
        MRAMSZ = $ - MRAM;

        region {} stack [size=0x500]; /* stack of at least 0x500 bytes */
        STACKTOP = 0x10000; /* stack pointer reset value */
    } o;
} p [size=0x10000];
```

B. FILENAME VERSION.H

```
extern void version(void);
```

C. FILENAME VERSION.C

```
void version(void);

/*****
void version(void)
{
    printf(
        "\nControl program for the Space Shuttle Vibro-acoustic Experiment. n.r.
        Version 6.19   April 14, 1989 n.r");
}
```

D. FILENAME VIBRO.H

```
/* vibro.h */

#define TRUE      0xff
#define FALSE     0x00
#define EXPERIMENTOK 0x11 /* As a parameter to menu(), this true flag
                           permits the experiment to be run. */

#define SELECT    0x1      /* Select appropriate power relay. */
#define ASCII     0        /* Used as a parameter to showbubbuff(). */
#define HEX       1        /* Used as a parameter to showbubbuff(). */
#define NULL      0x00 /* The following are ASCII definitions. */
#define BELL      0x07
#define BS        0x08
#define CTRLS     0x13     /* Permits output to be halted and restarted. */
#define CTRLY     0x19     /* Permits the menu() program to be entered
                           recursively anytime console I/O takes place.
                           Only one recursive call at a time is supported. */

#define SPACE     0x20
#define DELETE    0x7f

#define STANDBY   0x01      /* These are masks for the SSDR commands and. */
#define SWEEP     0x02      /* status codes. */
#define SCROLL    0x04
#define LAUNCH    0x08
#define RECORD    0x10
#define PLAYBACK  0x20
#define OPCOMP     0x40
#define NORMOP     0x80

#define TRIES     3        /* Number of times to try something before giving up. */

#define BLOCKS_PER_PAGE 2
                           /* The number of data blocks per page
                           of bubble memory. */
#define RECORD_DELAY 12    /* The number of hours to wait after initiating
                           RECORD mode before daring to restart it. */

/* The following constants are used by the routine adtoint() to convert
   values read by the A/D converter into the corresponding real-world units. */
#define MULT_TEMP 1960784L /* uK per unit on the A/D converter. */
```

```

#define MULT_10V 4862745L /* 10E-2V per unit on the A/D converter. */
#define MULT_20V 9725490L /* 10E-2V per unit on the A/D converter. */

#define MIN_VOLTAGE_10 850 /* 8.50 V is the minimum permissible
                           voltage on the 10V bus. The constant
                           represents this in units of 1E-2 V. */
#define MIN_OPERATING_TEMP 283 /* The bubble memories should not be operated
                                if the temperature falls below 10 degrees C
                                or 283 K. */
#define MIN_DESIRABLE_TEMP 285 /* The heaters should be on if the temperature
                                is below 285 K. */
#define MAX_DESIRABLE_TEMP 287 /* The heaters should be off if the temperature
                                is above 287 K. */

#define BUBDATA 0x40 /* I/O port for the controller's bubble memory. */
#define BUBCTRL 0x41 /* Control and status port for the BMC. */

/* The following codes are commands to the bubble memory controller. */
#define BABORT 0x19
#define BINIT 0x11
#define BIFORESET 0x10
#define BHRBLREG 0x16 /* Write boot loop register. */
#define BREAD 0x12
#define BWRITE 0x13
#define BLDPARAM 0x0b /* Load parametric registers. */

#define BTRIES 30000 /* Bubble commands should be written this */
                   /* many times before giving up in disgust. */

/* The following are bubble memory controller status codes. */
#define BBUSY 0x80
#define BOPCOMPLETE 0x40
#define BFAIL 0x20
#define BTIMING 0x02
#define BFIFO 0x01

#define BBUSYBIT 7 /* These constants specify which bit in the */
#define BOPCOMPLETEBIT 6 /* BMC status bit is used for which purpose. */
#define BFAILBIT 5
#define BFIFOBIT 0

#define BNEVER_READY 0
#define BXFER_GOOD 1
#define BXFER_BAD 2
#define PAGELNGTH 64 /* The number of bytes in a page of bubble memory. */
#define MAXPAGE 8191 /* Greatest valid bubble memory page number. */

#define ADPOINTS 10 /* The number of analog quantities to be converted to
                   decimal. */
#define STRLEN 7 /* Number of characters to allow for integer
                characters, including a null terminator. */
#define HSTRLEN 2 /* Number of characters to allow for hexadecimal
                characters*/
#define HEXINTSTRLEN 4 /* Number of characters in a hexadecimal word. */
#define DUMPWIDTH 16 /* Number of bytes in a line of a memory dump. */

```

```

/* Bit definitions for port C of NSC810 #1. (Base address is 0x00.)
  Bit #   MEANING
  5       Spare output.
  4       Power strobe output (Active high).
  3       One if no terminal is connected to the RS-232C port.
          Zero if a terminal is connected.
  2       Barometric pressure drop detection after launch
          (active high).
  1       Vibration detection at launch(active high).
  0       Matched filter detection of Auxiliary Power Unit (APU)
          prior to launch (active high).
*/

#define TERMON   0x08   /* Points to the terminal connection line in NSC810 #1,
                        Port C, Pin 3. It is zero when the terminal is
                        connected.*/

#define BARO_ON  0x04   /* Barometric pressure drop line. */
#define VIB_ON   0x02   /* Vibration detection line. */
#define APU_ON   0x01   /* APU detection line. */

/* Bit definitions for port C of NSC810 #2. (Base address is 0x20.)
  Bit #   Meaning
  5       RESET* line for the bubble memory. This line should be
          zero whenever power is applied to or removed from the
          bubble memory. It is one normally. The purpose of
          making it zero during power switching is to avoid havin
          to meet the strict requirements for power rise and fall
          times which would be necessary otherwise.
  4       Power line for the bubble memory. This line is a one
          to apply power; a zero to remove it.
  3       End of analog to digital conversion. (Active high?)
  2       Spare input.
  1       Spare input.
  0       Heater control output (active high).
*/

#define READC1   0x02   /* Points to the NSC810 #1, Port C, R/W register. */
#define BCLRC1   0x0a   /* Points to the NSC810 #1, Port C, Clear register. */
#define BSETC1   0x0e   /* Points to the NSC810 #1, Port C, Set register. */
#define BCLRC2   0x2a   /* Points to the NSC810 #2, Port C, Clear register. */
#define BSETC2   0x2e   /* Points to the NSC810 #2, Port C, Set register. */
#define PHRSTROBE 0x10  /* Points to the power board relay strobing line. To
                        turn on a peripheral, you must strobe this line high
                        for PHRDELAY * 10 ms. This line is NSC810 #1,
                        Port C, Pin 4. */

/* These are port addresses for the A/D converter. Character strings which
   identify these are defined in file "global.c". Be sure that changes
   in one place are matched in the other. */
#define VOLT0    0x80   /* Voltage from +20 V bus. */
#define VOLT1    0x81   /* Voltage from -20 V bus. */
#define VOLT2    0x82   /* Voltage from +10 V bus. */
#define TEMP0    0x83   /* Temperature from shelf above BMC. */
#define TEMP1    0x84   /* Temperature from underside of speaker. */
#define TEMP2    0x85   /* Temperature from shelf above battery. */
#define TEMP3    0x86   /* Temperature from batteries. */
#define TEMP4    0x87   /* Temperature from controller's backplane. */
#define TEMP5    0x88   /* Temperature from card 8 of BMC. */
#define TEMP6    0x89   /* Temperature from card 9 of BMC. */

```

```

#define PHRDELAY 2 /* The number of 10 ms units that the power board
                    strobe should be applied to turn on a relay. */
#define BUBRST 0x20 /* Points to the RESET* line in NSC810 #2, Port C,
                    Pin 5. */
#define BUBPHR 0x10 /* Points to the bubble power line in NSC810 #2,
                    Port C, Pin 4. */
#define BUBDELAY 5 /* Number of 10 ms units to wait when operating the
                    bubble memory. */

#define MDR1 0x07 /* See the documentation for a description of the */
#define DDRA1 0x04 /* use of these ports. */
#define DDRB1 0x05
#define DDRC1 0x06
#define TM01 0x18
#define TOLB1 0x10
#define TOHB1 0x11
#define START01 0x15
#define MDR2 0x27
#define DDRA2 0x24
#define DDRB2 0x25
#define DDRC2 0x26
#define TM02 0x38
#define TOLB2 0x30
#define TOHB2 0x31
#define START02 0x35

#define PRDATA 0xc0 /* Port number for data from RS-232C interface. */
#define PRCTRL 0xe0 /* Port number for control information from RS-232C
                    interface. */
#define PRTOURDY 0x01 /* Bit zero of the PRCTRL byte is a one if the printer
                    is ready to accept data and zero otherwise. */
#define PRTRDY 0x02 /* Bit one of the PRCTRL byte is a one if there is
                    data to be read and zero otherwise. */
/* Bit meanings for the power status byte at address POWERIN.
   Bit #   Meaning
   5       1 if heater circuit is off; 0 if it's on.
   4       1 if matched filter (APU detection) circuit is off;
           0 if it's on.
   3       1 if analog to digital converter (A/D) circuit is off;
           0 if it's on.
   2       1 if voltage controlled oscillator (VCO) is off;
           0 if it's on.
   1       1 if solid state data recorder (SSDR) is off;
           0 if it's on.

The same bit assignments apply to the power command byte at address POWEROUT,
but the bits have a different meaning. A one in bits 1-5 is used to select the
corresponding relay. A zero is used to cause that relay to be ignored.
A one in bit zero causes the selected relays to be switched on. A zero in
bit zero causes the selected relays to be switched off.
*/

#define PHR_RELAYS 5 /* The number of power relay switches. */
#define POWEROUT 0x01 /* Port address for power control board commands. */

```

```

#define POWERIN      0x21    /* Port address for power control board status. */
#define SSDROUT      0x00    /* Port address for SSDR commands. */
#define SSDRIN       0x01    /* Port address for SSDR status. */
#define SSDROFF      0x02    /* The following are commands for applying or removing*/
#define SSDRON       0x03    /* power.*/
#define VCOFF        0x04
#define VCOON        0x05
#define ADOFF        0x08
#define ADON         0x09
#define MATOFF       0x10
#define MATFON       0x11
#define HEATOFF      0x20
#define HEATON       0x21
#define ONBIT        0x01    /* The lowermost bit of a power command is 1 to turn
                             power on, 0 to turn it off. */
#define NOPOWER      0xC1    /* Mask for upper 2 bits and bit 0 in power. These
                             bits have no meaning when you examine the power
                             board's status.*/

/* These are event codes used for logging events. */
/* Dont' alter these codes without adjusting show_event() accordingly. */
/* A prefix C means COMMAND ISSUED.
   A prefix CF means COMMAND FAILED.
   A prefix CS means COMMAND SUCCEEDED.
   A prefix D means SOMETHING WAS DETECTED OR DONE. */
#define INITIALIZE   0    /* Start with aplomb. */
#define CSHEEP       1    /* SSDR was commanded to enter SHEEP mode. */
#define CSSHEEP      2    /* SSDR accepted a SHEEP command. */
#define CFSHEEP      3    /* SSDR wouldn't accept a SHEEP command. */
#define DSHEEP       4    /* The sweep was completed successfully. */
#define DAPUON       5    /* The auxiliary power unit was detected ON. */
#define CSCROLL      6    /* SSDR was commanded to enter SCROLL mode. */
#define CSSCROLL 7    /* SSDR accepted a SHEEP command. */
#define CFSCROLL 8    /* SSDR wouldn't accept a SCROLL command. */
#define DLAUNCH      9    /* A launch was detected. */
#define CLAUNCH     10    /* SSDR was commanded to enter LAUNCH mode. */
#define CSLAUNCH 11    /* SSDR accepted a LAUNCH command. */
#define CFLAUNCH 12    /* SSDR wouldn't accept a LAUNCH command. */
#define DPRESSURE    13    /* The pressure switch detected a pressure drop.*/
#define DNOPCOMP     14    /* SSDR didn't report completion in the
                             allotted time. */
#define DOPCOMP      15    /* SSDR completed its SHEEP or LAUNCH mode. */
#define DABORT       16    /* We think the mission was aborted. */
#define CONSSDR      17    /* The SSDR power on command was issued. */
#define CSONSSDR 18    /* The SSDR power on command succeeded. */
#define CFONSSDR 19    /* The SSDR power on command failed. */
#define COFFSSDR     20    /* The SSDR power off command was issued. */
#define CSOFFSSDR    21    /* The SSDR power off command succeeded. */
#define CFOFFSSDR    22    /* The SSDR power off command failed. */
#define COFFVCO      23    /* The VCO power off command was issued. */
#define CSOFFVCO 24    /* The VCO power off command succeeded. */
#define CFOFFVCO 25    /* The VCO power off command failed. */
#define CONVCO       26    /* The VCO power on command was issued. */
#define CSONVCO      27    /* The VCO power on command succeeded. */
#define CFONVCO      28    /* The VCO power on command failed. */
#define COFFAD       29    /* The AD power off command was issued. */
#define CSOFFAD      30    /* The AD power off command succeeded. */

```

```

#define CPOFFAD      31 /* The AD power off command failed. */
#define CONAD        32 /* The AD power on command was issued. */
#define CSONAD        33 /* The AD power on command succeeded. */
#define CFONAD        34 /* The AD power on command failed. */
#define COFFMATF      35 /* The MATF power off command was issued. */
#define CSOFFMATF      36 /* The MATF power off command succeeded. */
#define CPOFFMATF      37 /* The MATF power off command failed. */
#define CONMATF        38 /* The MATF power on command was issued. */
#define CSONMATF 39 /* The MATF power on command succeeded. */
#define CFONMATF 40 /* The MATF power on command failed. */
#define COFFHEAT      41 /* The HEAT power off command was issued. */
#define CSOFFHEAT      42 /* The HEAT power off command succeeded. */
#define CPOFFHEAT      43 /* The HEAT power off command failed. */
#define CONHEAT        44 /* The HEAT power on command was issued. */
#define CSONHEAT 45 /* The HEAT power on command succeeded. */
#define CFONHEAT 46 /* The HEAT power on command failed. */
#define READAD        47 /* We read the A/D's. */
#define TERMINATE      48 /* Finish gracefully. */
#define DUSERNOAPU      49 /* The user terminated the wait for the APUs. */
#define INVALIDCOMMAND 50 /* This code is regarded as invalid, and should
                           never occur. It is provided to help in
                           debugging the software. */
#define PRIORLAUNCH    51 /* If power is restored after the launch has already
                           begun, then this mission status is assigned. */

#define CSRECORD 52 /* The RECORD mode command succeeded. */
#define CFRECORD 53 /* The RECORD mode command failed. */

/* Various constants used for setting the parametric registers. */
#define BBLKLNM 0x10 /* Block length register MSB. 64 bytes/page. */
#define BBLKLNL 0x01 /* Block length register LSB. 1 page/transfer. */
#define BMBMSEL 0x00 /* Bubble memory select (BMB). Only 1 module
                     connected. */
#define BENREG 0x20 /* Enable register. Polling mode. */

#define THOUSANDTHS 0x60 /* The ports for reading the date and time. */
#define HUNDREDTHS 0x61
#define SECONDS 0x62
#define MINUTES 0x63
#define HOURS 0x64
#define WEEKDAY 0x65
#define DATE 0x66
#define MONTH 0x67

struct datetime { /* This structure contains binary coded */
    char month; /* decimal data as defined for the National */
    char date; /* Semiconductor MM58167A Microprocessor */
    char hour; /* Real Time Clock. */
    char minute;
    char second;
    char hundredths;
    char thousandths;
};

struct idatetime { /* This structure contains the same */
    int imonth; /* information as the datetime structure, but */
    int idate; /* in integer format. clockint() takes care */

```

```

int ihours;          /* of converting from BCD to integer format. */
int iminute;
int isecnd;
int ihundredths;
int ithousandths;
},

/* This structure describes the uses of the bits in the power relay control
port. */
struct power_port_fmt {
    char    :2;          /* Upper two bits are not used. */
    char    heater:1;     /* Bit 5 - designates the heater circuit. */
    char    matched_filter:1; /* Bit 4 - designates the matched filter. */
    char    a_to_d:1;     /* Bit 3 - designates the A/D circuit. */
    char    vco:1;        /* Bit 2 - designates the VCO. */
    char    ssdr:1;       /* Bit 1 - designates the SSDR. */
    char    relays_on:1;   /* Bit 0 - 1 to turn relays on,
                           0 otherwise. */
},

/* This structure describes data stored in page zero of the controller's
bubble memory. */
struct page0data {
    /* A template for data in page zero of the
    controller's bubble memory. */
    char sweepstarted; /* FALSE if sweep not yet begun.
                       TRUE if sweep has been started once. */
    char launchdone; /* FALSE if launch has not yet been detected.
                    TRUE if launch has been detected. */
    int page; /* Number of next page available for
             log data.*/
    char halfpage; /* 0 if top half of next available page is
                  empty, 1 otherwise. */
    char full_experiment; /* TRUE if the full experiment is to be performed,
                        FALSE otherwise. */
    /* We need to record the date and time when RECORD
    mode was last initiated if we are not performing
    the full experiment. */
    struct datetime RECORD_start_time;
},

/* This structure describes data stored in every block of a page in
the controller's
bubble memory, with the exception of page zero. */
struct log_data {
    /* A template for logged data. */
    struct datetime clock; /* Time and date of recorded data. */
    char event; /* A coded event. See #define section for
               codes. */
    char atod[ADPOINTS]; /* Coded A/D readings. Codes not yet defined. */
},

/* This structure has BLOCKS_PER_PAGE log_data structures in it. */
struct full_log_page {
    struct log_data half_page[BLOCKS_PER_PAGE];
},

```

```

enum pwr_cmd_modifiers (/* These codes can be given to cmdlog() for */
    issued      = 0,    /* processing. They show whether a power relay */
    succeeded    = 1,    /* command was merely issued, or succeeded or */
    failed       = 2     /* failed. */
);

```

E. FILENAME BUBBLE.H

```

/* This file contains global prototype declarations for the functions in
"bubble.c". */

```

```

extern void bpageset(int page);
extern void bubcmdmenu(void);
extern char bubinit(void);
extern char bubio(char command,int page,char *buffer);
extern void bubmenu(void);
extern char bub_on(void);
extern void bub_off(void);
extern char issububcmd(char command);
extern void rdstatreg(void);
extern void showbubbuff(char buffer[],char mode);
extern void testpattern(char buffer[]);

```

F. FILENAME BUBBLE.C

```

/* bubble.c */

#include "bubrw.h"
#include "vibro.h"
#include "convert.h"
#include "expmnt.h"
#include "inout.h"
#include "delay.h"
#include "newio.h"
#include "global.h"

void bpageset(int page);
void bubcmdmenu(void);
char bubinit(void);
char bubio(char command,int page,char *buffer);
void bubmenu(void);
char bub_on(void);    /* turn on power to the bubble card */
void bub_off(void);   /* turn off the power to the bubble card */
char issububcmd(char command);
void rdstatreg(void);
void showbubbuff(char buffer[],char mode); /* display the bubble buffer. */
void testpattern(char buffer[]); /* sets whole bubble
                                buffer to character of users choice*/

/*****
/* See the bubble memory manual regarding the setting of the parametric

```

```

    registers, which is what this function does. */
void bpageset(int page)
{
    output(BUBCTRL,BLDPARM); /* signal to BMC next 5 bytes to data port
                               are for parametric registers */
    output(BUBDATA,0x01);    /* one page to transfer BLR_LSB */
    output(BUBDATA,0x10);    /* one FSA channel BLR_MSB */
    output(BUBDATA,0x20);    /* Enable register-enable read corrected data */
    /* Mask off lower byte of page number. */
    output(BUBDATA,page & 0x00ff);
    /* Mask off higher byte of page number, with a zero MBM. */
    output(BUBDATA,(page >> 8) & 0x001f);
}

/*****
/* Select from a menu, and issue, a command to the bubble memory. */
void bubcmdmenu(void)
{
    char    data;
    static int  command[] = {
        BABORT, BLDPARM, BINIT, BFIFORESET,
        NULL, BHRBLREG
    };
    while (TRUE){
        printf("Select a command to be issued to the bubble memory:n r
A Abort  B Load parametric registers  C Initialize  D FIFO Resetn r
E Transfer 40 bytes of 0xff.           F Write bootloop register n r
Z Return to previous menu. n r");
        data = tolower(termin());
        printf("%c n r",data);

        if (data == 'z') return;

        /* Issue bubble transfer command. */
        if (data == 'e') {
            if (bubxfer())
                printf("Transfer succeeded. n r");
            else
                printf("Transfer failed. n r");
            continue;
        }

        /* Initialize parametric registers for page zero */
        if (data == 'b') {
            bpageset(0);
            continue;
        }

        /* Check for other valid responses */
        if (data < 'a' || data > 'f') {
            printf("Use a valid letter please. n r");
            continue;
        }

        /* Issue the command indexed by the letter */
        if (issububcmd(command[data-'a']))
            printf("Command succeeded. n r");
    }
}

```

```

        else
            printf("Command failed. n r");
    }
}

/*****
/* This routine initializes the controller card's bubble memory.
   Return FALSE if unsuccessful; TRUE otherwise.
   After power is applied to the bubble memory, call this routine.
   It implements the flow chart on pp. 4.9-4.9b in BPK 5V75A Prototyping Kit
   User's Manual from Intel. */
char bubinit(void)
{
    /* Clear the bubble memory registers. */
    if (!issububcmd(BABORT)) {
        printf("ABORT command failed. n r");
        return(FALSE);
    }
    delay(BUBDELAY);          /* Delay BUBDELAY * 10 ms */
    bpageset(0);              /* Load the parametric registers of
                               the bubble memory. */
    if (!issububcmd(BINIT)) { /* Initialize bubble memory for use. */
        printf("INITIALIZE command failed. n r");
        return(FALSE);
    }
    if (!issububcmd(BFIFORESET)) { /* Reset FIFO buffer. */
        printf("FIFO RESET command failed in bubinit(). n r");
        return(FALSE);
    }
    if (!bubxfer()) {         /* Write 40 0xff characters to the
                               bubble memory controller. */
        printf("40 byte transfer failed. Status: ");
        rdstatreg();
        return(FALSE);
    }
    if (!issububcmd(BHRLREG)) /* Put boot loop memory map into BMC. */
        return(FALSE);
    return(TRUE);            /* If you got this far, everything
                               worked. */
}

/*****
/* Perform normal input from or output to the bubble memory
   controller. */
char bubio(char command,int page,char *buffer)
/* "command" can be BREAD to read; BWRITE to write. */
/* "page" is a bubble memory page number, from 0 to 8192. */
/* "buffer" is a pointer to a buffer of length PAGELENGTH. */
{
    int j; /* Counters. */

    /* Do not operate the bubble memory if the temperature is below
       MIN_OPERATING_TEMP. */
    if (colder_than(MIN_OPERATING_TEMP))
        return(FALSE);

```

```

bub_on();
if (!bubinit()) {
    bub_off();
    return(FALSE);
}
bpageset(page); /* Set parametric registers for the desired page. */
if (command == BREAD) {
    bubread(buffer);
} else if (command == BWRITE) {
    bubwrite(buffer);
}
/* Wait for the BMC to finish emptying the FIFO buffer and return
OPCOMPLETE. */
for (j=0;j<BTRIES;++j) {
    if (input(BUBCTRL) & BOPCOMPLETE)
        break;
}
if (j >= BTRIES) {
    bub_off();
    printf("Couldn't get an OPCOMPLETE from BMC. Status: ");
    rdstatreg();
    return(FALSE);
}
bub_off();
/* printf("OPCOMPLETE received from BMC.\n"); */
return(TRUE); /* If you got this far, the I/O worked! */
}

/*****
void bubmenu(void)
{
    char    data;
    static char success[] = {
        "Bubble was successfully initialized in bubmem().\n"
    };
    static char failure[] = {
        "Bubble couldn't be initialized in bubmenu().\n"
    };
    while (TRUE) {
        printf(
"A Turn bubble memory power on.\n"
B Turn bubble memory power off.\n"
C Initialize bubble memory for use (fully automatic).\n"
    Be sure to turn the bubble memory power on, first.\n"
D Issue one of a menu of commands to the bubble memory.\n"
E Enter data from Keyboard into buffer.\n"
F Show buffer contents in ASCII format.\n"
G Show buffer contents in hexadecimal format.\n"
H Copy buffer contents to bubble memory (write bubble memory).\n"
I Copy contents of bubble memory to buffer (read bubble memory).\n"
J Display contents of bubble memory status register.\n"
Z Return to previous menu.\n");

        data = tolower(termin());
        printf("%c\n",data);
        switch(data){

```

```

        case 'a':
            bub_on();
            break;
        case 'b':
            bub_off();
            break;
        case 'c':
            if (bubinit())
                printf(success);
            else
                printf(failure);
            break;
        case 'd':
            bubcmdmenu();
            break;
        case 'e':
            testpattern( tempbuffer );
            break;
        case 'f':
            showbubbuff( tempbuffer, ASCII );
            break;
        case 'g':
            showbubbuff( tempbuffer, HEX );
            break;
        case 'h':
            if (!bubio(BWRITE, getpageno(), tempbuffer))
                printf("Write Failed. n r");
            break;
        case 'i':
            if (!bubio(BREAD, getpageno(), tempbuffer))
                printf("Read Failed. n r");
            break;
        case 'j':
            rdstatreg();
            break;
        case 'z': case 'Z':
            return;
        default:
            printf("Use a valid letter, please. n r");
    }
}

/*****
char bub_on(void) /* turn on power to the bubble card */
{
    output(BCLRC2, BUBRST); /* Apply a reset to the bubble memory.*/
    output(BSETC2, BUBPHR); /* Apply power to the bubble memory.*/
    /* The following delays could be 100 ms (according to the bubble
       documentation) but did not work, so we used 300 ms. */
    delay(BUBDELAY); /* Wait BUBDELAY * 10 ms for a response. */
    output(BSETC2, BUBRST); /* Remove reset signal. */
    delay(BUBDELAY); /* Wait BUBDELAY * 10 ms for a response. */
}

```

```

/*****
void bub_off(void) /* turn off the power to the bubble card */
{
    issububcmd(BABORT); /* Issue the "abort" command to the bubble card. */
    output(BCLRC2,BUBRST); /* Apply a reset signal to the bubble memory
                           before switching the power off. */
    delay(BUBDELAY); /* Wait BUBDELAY * 10 ms for a response. */
    output(BCLRC2,BUBPMR); /* Remove power from the bubble memory. */
}

/*****
/* Issue a command to the bubble memory controller. */
char issububcmd(char command)
{
    int i;
    char status;
    i=0; /* Initialize this so it has a value even if BABORT
         is the command. */
    /* Don't issue a command until the BUSY bit goes away. */
    if (command != BABORT) {
        for (i=0;i < BTRIES;++i) {
            if ((input(BUBCTRL) & BBUSY)
                break;
        }
    }
    if (i >= BTRIES) {
        printf("Bubble controller stayed busy indefinitely in issububcmd().\n");
        Status: " );
        rdstatreg();
        return(FALSE);
    }
    output(BUBCTRL,command);
    /* Command is not accepted until busy bit goes to one. */
    for (i = 0;i < BTRIES;++i){
        status = input(BUBCTRL);
        if ((status & BBUSY) || (status & BOPCOMPLETE))
            break;
    }
    /* For all commands except RESET FIFO and WRITE BOOTLOOP REGISTERS,
       you must get a BUSY bit to consider that the command was accepted.
       However, an OPCOMPLETE is okay; if you get it, proceed. Note:
       this is not the way the documentation says to do this. It says
       you must get BUSY set first. However, that didn't seem to work. */
    if ((i >= BTRIES)
        && (command != BFORESET) && (command != BWRBLREG)
        && ( status & BOPCOMPLETE)) {
        printf("Bubble command %sh was not accepted. Status: ",
            ctoh(command));
        rdstatreg();
        return(FALSE);
    }
    /* Wait for the OPCOMPLETE status code. */
    for (i = 0;i < BTRIES;++i){
        if (input(BUBCTRL) & BOPCOMPLETE)
            break;
    }
    if (i >= BTRIES) {

```

```

        printf("OPCOMPLETE from BMC never occurred for command %sh. Status: ",
            ctoh(command));
        rdstatreg();
        return(FALSE);
    } else {
        return(TRUE);
    }
}

/*****
void rdstatreg(void)
{
    printf("%s\n",ctoh(input(BUBCTRL)));
}

/*****
void showbubbuff(char buffer[],char mode) /* display the bubble buffer.
    ASCII format tries to print each character as if it were a
    printable ASCII character.
    HEX format is the correct option to use if not all characters are
    printable. */
/* Valid values for "mode" are ASCII and HEX */
{
    int j;          /* Dump contents in an 8 by 8 array. */
    for (j=0;j<PAGELENGTH;j++) {
        if (mode == ASCII)
            printf("%c",buffer[j]);
        else {
            printf("%s ",ctoh(buffer[j]));
            if ((0 == (j + 1) % 8) && (j != 0))
                printf("\n");
        }
    }
    printf("\n");
}

/*****
void testpattern(char buffer[])
/* sets whole bubble buffer to character of users choice*/
{
    char    c;
    char    s[STRLEN]; /* Storage for itoa(). */
    int j;

    /* Make sure c has a value before checking its contents.*/
    c = '0';
    printf("Specify up to %s characters to stuff into the bubble.\n",
        itoa(PAGELENGTH,s));
    for (j=0;j<PAGELENGTH;j++) {
        if (c != 'r') {
            c = termin();
            if (c != 'r') {
                buffer[j] = c;
                printf("%c",c);
            } else
                buffer[j] = ' ';
        }
    }
}

```

```

    } else
        buffer[j] = ' ';
    }
    printf(" n r");
}

```

G. FILENAME BUBRW.H

```

extern char bubxfer(void);
extern char bubread(char *buffer);
extern char bubwrite(char *buffer);

```

H. FILENAME BUBRW.S

```

; bubrw.s

#define TRUE          0xff
#define FALSE        0x00

; The definitions which follow are from the file "vibro contrlr headers bmc.h".
; Since they are used by C source code, they are incompatible with assembly
; code. Thus they are copied here and all C comments have been converted
; to assembly language comments.

#define BUBDATA        0x40      ; I/O port for the controller's bubble memory. */
#define BUBCTRL        0x41      ; Control and status port for the BMC. */
; The following codes are commands to the bubble memory controller. */
#define B2BORT         0x19
#define BINIT          0x11
#define BFIFORESET     0x1D
#define BWRBLREG 0x16      ; Write boot loop register. */
#define BREAD          0x12
#define BWRITE         0x13
#define BLDPARAM       0x0b      ; Load parametric registers. */

#define BTRIES         30000      ; Bubble commands should be written this */
                                ; many times before giving up in disgust. */

; The following are bubble memory controller status codes. */
#define BBUSY          0x80
#define BOPCOMPLETE    0x40
#define BFAIL          0x20
#define BTIMING         0x02
#define BFIFO          0x01

#define BBUSYBIT       7          ; These constants specify which bit in the */
#define BOPCOMPLETEBIT 6          ; BMC status bit is used for which purpose. */
#define BFAILBIT       5
#define BFIFOBIT       0

#define BNEVER_READY   0
#define BXFER_GOOD     1
#define BXFER_BAD      2
#define PAGELENGTH     64        ; The number of bytes in a page of bubble memory. */
#define MAXPAGE        8191      ; Greatest valid bubble memory page number. */

```

```

;Implement in assembly code a C routine to permit a very rapid transfer
;of forty bytes of 0xff to the bubble memory controller during its
;initialization.
;char bubxfer(void)
;{
    export    bubxfer
    region    code
bubxfer:
    push     ix
    ld       ix,0           ; ix <-- sp
    add      ix,sp
;Issue a FIFO RESET command
    ld       a,$BFIFORESET  ; FIFO RESET command code.
    out      ($BUBCTRL),a
    ld       de,0xffff       ; Initialize a timeout counter.
fifors_busy:
                ; See if the command was accepted.
    in       a,($BUBCTRL)
    rla      ; Move busy bit into carry flag.
    jp       c,fiforst_accepted ; The busy bit is a 1 if the command
                                ; was accepted.
    dec      de
    xor      a               ; Clear register a.
    or       d               ; See if de is 0.
    or       e
    jp       nz,fifors_busy  ; Check for busy bit again, since timeout
                                ; not yet complete.
    ld       a,$FALSE
    jp       bxfer_exit      ; Timed out without succeeding, so
                                ; return with a FALSE condition code.
fiforst_accepted:
    ld       b,40            ; We need to transfer 40 bytes of 0xff
    ld       a,0xff
xfer: out     ($BUBDATA),a
    djnz     xfer
    in       a,($BUBCTRL)
; The transfer succeeded if you got an Op Complete code
; with the FIFO bit set, even with the timing bit (bit 1) set.
    and      $BTIMING        ; Zeroize the timing bit.
    cp       $BOPCOMPLETE | $BFIFO ; Do we have operation complete?
    jp       z,xfer_ok       ; Yes.
    ld       a,$FALSE
                                ; Unsuccessful transfer.
    jp       bxfer_exit
xfer_ok:
    ld       a,$TRUE         ; Successful transfer.
bxfer_exit:
    pop      ix
    ret
;}

;Implement in assembly code a C routine to permit very rapid input of
;a page of data from the bubble memory.
;char  bubread(char *buffer)
;{
    export    bubread
    import    issububcmd

```

```

        region    code
bubread:
#comment
    Register Usage:
        a    Scratch space.
        bc   Constant 1 for subtractions.
        de   Constant PAGELENGTH.
        hl   Constant BTRIES
        bc'  char *buffer.
#endcomment
    push     ix
    ld       ix,0           ; ix <-- sp
    add      ix,sp
    exx
    push     bc             ; Access alternate registers.
    push     bc             ; Save bc'
    ld       c,(ix+4)       ; bc' <-- char *buffer
    ld       b,(ix+5)
    exx
    ld       hl,$BFIFORESET ; Return to primary registers.
    push     hl             ; Issue the FIFO Reset command to the BMC.
    call     issububcmd
    pop      hl
    cp       $FALSE        ;Quit if this command didn't work.
    jp       nz,frok
    ld       a,$B$FER_BAD
    jp       exit
frok:
    ld       a,$BREAD       ; Issue the READ command to the BMC.
    out      ($BUBCTRL),a
    ld       hl,$BTRIES-1   ; Look for BUSY bit up to BTRIES times.
    ld       bc,1           ; Used for subsequent decrements.
read_status:
    in       a,($BUBCTRL)   ; Get status from BMC.
    bit     $BBUSYBIT,a     ; Was the command accepted?
    jr      nz,read         ; Yes, so read a block of data.
    bit     $BOPCOMPLETEBIT,a ; Not busy. Was operation complete?
    jr      nz,read         ; Yes, so read a block of data.
    bit     $BFAILBIT,a     ; Not busy, not done. Failed?
    jr      nz,timeoutl     ; Didn't fail. Don't know why. Allow a
    ; timeout.
    ld      a,$BNEVER_READY ; Did fail, so quit. Return function
    jp      exit            ; completion code in register a.
timeoutl:
    or      a               ; Reset the CARRY flag.
    sbc     hl,bc           ; Have we looked for a BUSY signal BTRIES
    ; times yet?
    jr      nc,read_status  ; No, so try again.
    ld      a,$BNEVER_READY ; Yes, so we timed out. Quit and return
    ; function completion code in register a.
    jp      exit
read:
    ld      de,$PAGELENGTH-1 ; Prepare to read PAGELENGTH bytes from BMC.
read_byte:
    ld      hl,$BTRIES-1    ; Prepare to check FIFO bit BTRIES times.
check_fifo:
    in      a,($BUBCTRL)    ; Get status byte from BMC.
    bit     $BFIFOBIT,a     ; Is the FIFO bit set, i.e. FIFO ready?

```

```

        jr      nz,getbyte      ; Yes, so read a byte.
        or      a               ; Reset the CARRY flag.
        sbc     hl,bc           ; No, so try again up to BTRIES times.
        jr      nc,check_fifo
        ld      a,$B_XFER_BAD   ; Never got a FIFO ready, so quit.
        jp      exit
getbyte:
        in      a,($BUBDATA)    ; Read a byte from the BMC.
        exx
        ld      (bc),a         ; Place the byte in the buffer.
        inc     bc              ; Point to the next position in the buffer.
        exx
        or      a               ; Reset the CARRY flag.
        ex      de,hl           ; Put contents of de in hl to permit use of sbc.
        sbc     hl,bc           ; Have we read all the bytes yet?
        ex      de,hl           ; Restore usual contents to de and hl.
        jr      nc,read_byte    ; No, so get another one.
        ld      a,$B_XFER_GOOD  ; Yes, so quit. Return function completion
                                ; code in register a.
exit:
        exx
        pop     bc              ; Restore alternate registers.
        exx
        pop     ix              ; Restore ix register.
        ret

;)
```

;Implement in assembly code a C routine to permit very rapid output of
;a page of data to the bubble memory.

```

;char  bubwrite(char *buffer)
```

```

;{
```

```

    export  bubwrite
    import  issububcmd
    region  code
```

```

bubwrite:
```

```

#comment
```

```

    Register usage:
```

```

        a      Scratch space.
        bc     Constant 1 for subtractions.
        de     Constant PAGEDLENGTH.
        hl     Constant BTRIES
        bc'    char *buffer.
```

```

#endcomment
```

```

    push     ix
    ld      ix,0                ; ix <-- sp
    add     ix,sp
    exx
    push     bc                  ; Access alternate registers.
                                ; Save bc'
    ld      c,(ix+4)            ; bc' <-- char *buffer
    ld      b,(ix+5)
    exx
    ld      hl,$B_FIFORESET     ; Issue the FIFO Reset command to the BMC.
    push     hl
    call     issububcmd
    pop     hl
    cp      $FALSE              ;Quit if this command didn't work.
```

```

jp      nz,frok2
ld      a,$BXFER_BAD
jp      exit2

frok2:
ld      a,$BWRITE          ; Issue the WRITE command to the BMC.
out     ($BUBCTRL),a
ld      hl,$BTRIES-1       ; Look for BUSY bit up to BTRIES times.
ld      bc,1               ; Used for subsequent decrements.

write_status:
in      a,($BUBCTRL)       ; Get status from BMC.
bit     $BBUSYBIT,a        ; Was the command accepted?
jr      nz,write           ; Yes, so write a block of data.
bit     $BOPCOMPLETEBIT,a  ; Not busy. Was operation complete?
jr      nz,write           ; Yes, so write a block of data.
bit     $BFAILBIT,a       ; Not busy, not done. Failed?
jr      nz,timeout2       ; Didn't fail. Don't know why. Allow a
                           ; timeout.
ld      a,$BNEVER_READY    ; Did fail, so quit. Return function
jp      exit2              ; completion code in register a.

timeout2:
or      a                  ; Reset the CARRY flag.
sbc     hl,bc              ; Have we looked for a BUSY signal BTRIES
                           ; times yet?
jr      nc,write_status    ; No, so try again.
ld      a,$BNEVER_READY    ; Yes, so we timed out. Quit and return
                           ; function completion code in register a.
jp      exit2

write:
ld      de,$PAGELENGTH-1  ; Prepare to write PAGELENGTH bytes to BMC.

write_byte:
ld      hl,$BTRIES-1       ; Prepare to check FIFO bit BTRIES times.

check_fifo2:
in      a,($BUBCTRL)       ; Get status byte from BMC.
bit     $BFIFOBIT,a        ; Is the FIFO bit set, i.e. FIFO ready?
jr      nz,putbyte         ; Yes, so read a byte.
or      a                  ; Reset the CARRY flag.
sbc     hl,bc              ; No, so try again up to BTRIES times.
jr      nc,check_fifo2     ; Never got a FIFO ready, so quit.
ld      a,$BXFER_BAD
jp      exit

putbyte:
exx
ld      a,(bc)             ; Get the byte from the buffer.
out     ($BUBDATA),a       ; Write a byte to the BMC.
inc     bc                 ; Point to the next position in the buffer.
exx
or      a                  ; Reset the CARRY flag.
ex      de,hl              ; Put de into hl to permit use of sbc.
sbc     hl,bc              ; Have we read all the bytes yet?
ex      de,hl              ; Restore usual contents to de and hl
jr      nc,write_byte      ; No, so get another one.
ld      a,$BXFER_GOOD      ; Yes, so quit. Return function completion
                           ; code in register a.

exit2:
exx
pop     bc                 ; Restore alternate registers.
exx

```

```

        pop        ix            ; Restore ix register.
        ret

; }

```

I. FILENAME CLOCK.H

```

/* This file contains external prototyping declarations of all functions used
   in "clock.c". */

```

```

extern void clockint(struct datetime *clock, struct idatetime *iclock);
extern void clockread(struct datetime *your_clock);
extern char clockcompare(struct idatetime *clock1, struct idatetime *clock2);
extern void clockset(struct datetime *clock);
extern void clocksum(struct idatetime *result,
                    struct idatetime *clock1,
                    struct idatetime *clock2);
extern void dump_clock(struct datetime *clock);
extern void rtc(void);
extern void show_waketime(struct idatetime *waketime);
extern void testtimeout(void);
extern char timeout(int delaytime, int measure);

```

J. FILENAME CLOCK.C

```

/* clock.c */

```

```

#include "vibro.h"
#include "convert.h"
#include "inout.h"
#include "newio.h"
#include "global.h"

void clockint(struct datetime *clock, struct idatetime *iclock);
void clockread(struct datetime *your_clock);
char clockcompare(struct idatetime *clock1, struct idatetime *clock2);
void clockset(struct datetime *clock);
void clocksum(struct idatetime *result,
              struct idatetime *clock1,
              struct idatetime *clock2);
void dump_clock(struct datetime *clock);
void dump_iclock(struct idatetime *iclock);
void get_time(struct datetime *date_and_time);
void rtc(void);
void show_waketime(struct idatetime *waketime);
char *strcpy(char *s1, char *s2);
void testtimeout(void);
char timeout(int delaytime, int measure);

static char *months[] = {
    "** Invalid month **", "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
};

```

```

/*****/
/* Convert a datetime structure to an idatetime equivalent. This allows
   arithmetic to be performed on dates and times. */
void clockint(struct datetime *clock, struct idatetime *iclock)
{
    iclock->imonth   = bcd_int(clock->month);
    iclock->idate    = bcd_int(clock->date);
    iclock->ihour    = bcd_int(clock->hour);
    iclock->iminute  = bcd_int(clock->minute);
    iclock->isecond   = bcd_int(clock->second);
    iclock->ihundredths = bcd_int(clock->hundredths);
    iclock->ithousandths = bcd_int(clock->thousandths);
}

/*****/
/* This routine fills a clock structure with the current date and time. */
/* It will not worry about the hundredths and thousandths, but it will attempt
   to ensure that at least the seconds have not changed between the first
   and the last reads of the various clock registers. Thus the hundredths
   and thousandths should not be regarded as accurate, ever. */
void clockread(struct datetime *your_clock)
{
    int i;

    i = 0;
    do {
        your_clock->thousandths = input(THOUSANDTHS);
        your_clock->hundredths = input(HUNDREDTHS);
        your_clock->second      = input(SECONDS);
        your_clock->minute      = input(MINUTES);
        your_clock->hour        = input(HOURS);
        your_clock->date        = input(DATE);
        your_clock->month       = input(MONTH);
    } while (your_clock->second != input(SECONDS) && ++i <= 10 * TRIES);
}

/*****/
/* Compare two clock times. Return TRUE if the first is later than or
   equal to the second, FALSE otherwise. This routine ignores the
   hundredths and thousandths, since they are inaccurate. */
char clockcompare(struct idatetime *clock1, struct idatetime *clock2)
{
    int difference;

    difference = clock1->imonth - clock2->imonth;
    /* This logic allows you to decide January comes after December. */
    if ((difference + 12) % 12 < 6
        && difference != 0) return(TRUE);
    if (difference != 0) return(FALSE);
    if (clock1->idate < clock2->idate) return(FALSE);
    if (clock1->idate > clock2->idate) return(TRUE);
    if (clock1->ihour < clock2->ihour) return(FALSE);
    if (clock1->ihour > clock2->ihour) return(TRUE);
    if (clock1->iminute < clock2->iminute) return(FALSE);
    if (clock1->iminute > clock2->iminute) return(TRUE);
}

```

```

    if (clock1->isecnd < clock2->isecnd) return(FALSE);

    return(TRUE);
}

/*****
/* This routine sets the real time clock. */
void clockset(struct datetime *clock)
{
    get_time(clock);
    output(MONTH,clock->month);
    output(DATE,clock->date);
    output(HOURS,clock->hour);
    output(MINUTES,clock->minute);
    output(SECONDS,clock->second);
}

/*****
/* Find the sum of two calendar periods. */
void clocksum(struct idatetime *result,
              struct idatetime *clock1,
              struct idatetime *clock2)
{
    int maxdate;          /* The last valid date in the month. */

    result->isecnd = clock1->isecnd + clock2->isecnd;
    result->iminute = result->isecnd / 60;
    result->isecnd %= 60;
    result->iminute += clock1->iminute + clock2->iminute;
    result->ihour = result->iminute / 60;
    result->iminute %= 60;
    result->ihour += clock1->ihour + clock2->ihour;
    result->idate = result->ihour / 24;
    result->ihour %= 24;
    result->idate += clock1->idate + clock2->idate;
    result->imonth = 1 + (clock1->imonth + clock2->imonth - 1) % 12;
    maxdate = ((result->imonth == 4) || (result->imonth == 6)
               || (result->imonth == 9) || (result->imonth == 11)) ? 30 : 31;
    /* The real time clock makes no provision for leap year, so leap years
       are ignored in this program (sigh!) */
    maxdate = (result->imonth == 2) ? 28 : maxdate;
    result->imonth += (result->idate - 1) / maxdate;
    result->idate = 1 + (result->idate - 1) % maxdate;
    result->imonth = 1 + (result->imonth - 1) % 12;
}

/*****
/* Print a clock structure. */
void dump_clock(struct datetime *clock)
{
    int hour, minute, second, date, month;

    hour   = bcd_int(clock->hour);
    minute = bcd_int(clock->minute);
    second = bcd_int(clock->second);
    date   = bcd_int(clock->date);
    month  = bcd_int(clock->month);
}

```

```

printf("%02.2d:%02.2d:%02.2d %s %d\n",
    hour,minute,second,
    months[ month > 12 ? 0 : month ],
    date
);
}

/*****/
/* Print an iclock structure. */
void dump_iclock(struct idatetime *clock)
{
    printf("%02.2d:%02.2d:%02.2d %s %d\n",
        clock->ihour,clock->iminute,clock->isecond,
        months[ clock->imonth > 12 ? 0 : clock->imonth ],
        clock->idate
    );
}

/*****/
void get_time(struct datetime *date_and_time)
{
    int month, date, hour, minute, second, maxdate;
    static char cr[] = "n";

    while (TRUE) {
        printf("Month? (1-12) ");
        month = getint();
        if (month >= 1 && month <= 12)
            break;
        printf("Invalid month. Re-enter it.\n");
    }
    printf(cr);
    maxdate = (month == 4 || month == 6 || month == 9 || month == 11) ?
        30 : 31;
    maxdate = (month == 2) ? 28 : maxdate;
    while (TRUE) {
        printf("Day? (1-%d) ",maxdate);
        date = getint();
        if (date >= 1 && date <= maxdate)
            break;
        printf("\nInvalid date. Re-enter it.\n");
    }
    printf(cr);
    while (TRUE) {
        printf("Hour? (0-23) ");
        hour = getint();
        if (hour >= 0 && hour <= 23)
            break;
        printf("Invalid hour. Re-enter it.\n");
    }
    printf(cr);
    while (TRUE) {
        printf("Minute? (0-59) ");
        minute = getint();
        if (minute >= 0 && minute <= 59)
            break;

```

```

        printf("Invalid minute. Re-enter it. n r");
    }
    printf(cr);
    while (TRUE) {
        printf("Second? (0-59) ");
        second = getint();
        if (second >= 0 && second <= 59)
            break;
        printf("Invalid second. Re-enter it. n r");
    }
    printf(cr);
    date_and_time->month = int_bcd(month);
    date_and_time->date = int_bcd(date);
    date_and_time->hour = int_bcd(hour);
    date_and_time->minute = int_bcd(minute);
    date_and_time->second = int_bcd(second);
}

/*****
/* This routine is a menu-driven collection of routines for testing the
clock functions. */
void rtc(void)
{
    char data;

    while (TRUE) {
        printf(
"n rReal time clock functions. n r n r"
A Read Clock. n r
B Set clock. n r
C Test timeout() function. n r
Z Return to main menu. n r");

        data = tolower(termin());
        printf("%c n r", data);
        switch (data) {
            case 'a':
                clockread(&clock);
                dump_clock(&clock);
                break;
            case 'b':
                clockset(&clock);
                break;
            case 'c':
                testtimeout();
                break;
            case 'z':
                return;
            default:
                printf("Use a valid letter please. n r");
                break;
        }
    }
}

```

```

/*****
/* This routine displays the wake-up time. */
void show_waketime(struct idatetime *waketime)
{
    char s[STRLEN]; /* String for itoa() routine. */

    itoa(waketime->imonth,s);
    printf("Wake-up time is:  nMonth = %s ",s);
    itoa(waketime->idate,s);
    printf("Date = %s ",s);
    itoa(waketime->ihour,s);
    printf("Hour = %s ",s);
    itoa(waketime->iminute,s);
    printf("Minute = %s ",s);
    itoa(waketime->isecond,s);
    printf("Second = %s n",s);
}

/*****
/* This routine is used to test the timeout() function. */
void testtimeout(void)
{
    char    data;      /* A character entered from the keyboard. */
    units;   /* The units of delay. */
    int delay; /* The number of units of delay. */

    while (TRUE) {
        printf("Test of timeout() function. n n n
Specify time units for delay: n n n
A Hours n n
B Minutes n n
C Seconds n n
Z Return to previous menu. n n");

        data = tolower(termin());
        printf("%c n",data);
        switch (data) {
            case 'a':
                units = HOURS;
                break;
            case 'b':
                units = MINUTES;
                break;
            case 'c':
                units = SECONDS;
                break;
            case 'z':
                return;
                break;
            default:
                printf("Use a valid letter please. n n");
                break;
        }

        printf("n nHow many units of delay do you want? n n");
        delay = getint();
        printf("n nStarting delay: n n");
    }
}

```

```

        clockread(&clock);
        dump_clock(&clock);
        timeout(delay,units);
        while( !timeout(NULL,NULL) );
        printf("Delay complete. n r");
        printf("%c",BELL);
        clockread(&clock);
        dump_clock(&clock);
    }
}

/*****
/* This routine is used to initiate a timeout sequence, and to test for
completion. To set the desired delay time, the parameter "delay"
should be non-zero. To test for completion, "delay" should be zero (NULL).
When setting the delay time, the function always returns TRUE. When
testing for completion, it returns TRUE if the time has elapsed, FALSE
otherwise. */
char timeout(int delaytime,int measure)
/* "delaytime" is the length of the timeout. */
/* "measure" is the unit of measure of time. This can be
MONTH, DATE, HOURS, MINUTES, or SECONDS. */
{
    static struct datetime timenow;
    static struct idatetime itimenow, waittime;

    /* Allow the user to interrupt by use of CTRL characters. */
    allow_ctrl_interrupts();

    clockread(&timenow);
    clockint(&timenow,&itimenow);
    if (delaytime == NULL) { /* If delaytime == NULL, then check to
                             see if timeout period is over. */
        return(clockcompare(&itimenow,&waketime));
    } else { /* Otherwise, set the wakeup time. */
        waittime.imonth = waittime.idate = waittime.ihour
            = waittime.iminute = waittime.isccond = 0;
        switch(measure) {
            case MONTH:
                waittime.imonth = delaytime;
                break;
            case DATE:
                waittime.idate = delaytime;
                break;
            case HOURS:
                waittime.ihour = delaytime;
                break;
            case MINUTES:
                waittime.iminute = delaytime;
                break;
            case SECONDS:
                waittime.isccond = delaytime;
                break;
        }
        clocksum(&waketime,&itimenow,&waittime);
        show_waketime(&waketime);
    }
}

```

```

        return(TRUE);
    }
}

```

K. FILENAME CONVERT.H

```

/* This file contains external prototyping declarations for all functions
in "convert.c". */

```

```

extern char atoh(char *ascii);
extern unsigned int atohexint(char ascii[]);
extern int atoi(char *s);
extern char *bcd_asc(char bcd);
extern int bcd_int(char bcd);
extern char *ctoh(char byte);
extern char int_bcd(int decimal);
extern char *itoa(int n, char s[]);
extern char tolower(int c);
extern char *uitoh(unsigned int word);

```

L. FILENAME CONVERT.C

```

/* convert.c */

```

```

#include "vibro.h"
#include "inout.h"
#include "global.h"

```

```

char atoh(char *ascii);
unsigned int atohexint(char ascii[]);
int atoi(char *s);
char *bcd_asc(char bcd);
int bcd_int(char bcd);
char *ctoh(char byte);
char int_bcd(int decimal);
char *itoa(int n, char s[]);
char tolower(int c);
char *uitoh(unsigned int word);

```

```

/*****
/* This routine converts a two-byte ASCII string representing a valid
   hexadecimal byte into a single hexadecimal byte. */
*****/
/*****
char atoh(char *ascii)
    /* "ascii" is a string representing a hexadecimal byte. */
{
    int i;
    char    result; /* The hexadecimal byte after conversion. */

    result = 0;
    for (i=0; i < HSTRLEN && ascii[i] != NULL; ++i) {
        result *= 16;
        if ( '0' <= ascii[i] && '9' >= ascii[i])
            result += ascii[i] - '0';
    }
}

```

```

        else if ('a' <= ascii[i] && 'f' >= ascii[i])
            result += 10 + ascii[i] - 'a';
    }
    return(result);
}

/*****
/* This routine converts a four-byte ASCII string representing a valid
   hexadecimal word into a single unsigned integer. */
*****/
unsigned int atohexint(char ascii[])
{
    int i;
    unsigned int result; /* The hexadecimal word after conversion. */

    result = 0;
    for (i=0; i < HEXINTSTRLLEN && ascii[i] != NULL; ++i) {
        result *= 16;
        if ('0' <= ascii[i] && '9' >= ascii[i])
            result += ascii[i] - '0';
        else if ('a' <= ascii[i] && 'f' >= ascii[i])
            result += 10 + ascii[i] - 'a';
    }
    return(result);
}

/*****
int atoi(char *s)      /* convert string to integer */
{
    static int n, sign;
    sign = 1;
    n = 0;
    switch (*s) {
        case '-': sign = -1;
        case '+': ++sign;
    }
    while (*s >= '0' && *s <= '9') n = 10 * n + *s++ - '0';
    return(sign * n);
}

/*****
/* Convert a byte of binary coded decimal data to character string format. */
/* No check is made to ensure that input data really IS in BCD format. */
char *bcd_asc(char bcd) /* Tested March 16, 1987 */
{
    static char ascii[3];
    int bcdint;

    bcdint = 0x00ff & ((int) bcd); /* Convert to integer. */
    /* If the tens digit is a zero, put a blank in its place;
       otherwise, put an ASCII digit there. */
    ascii[0] = (0xf0 & bcdint) ?
        (0x30 | (bcdint >> 4)) : ' ';
    ascii[1] = 0x30 | (bcdint & 0xf); /* Get the units digit. */
    ascii[2] = NULL; /* Terminate the string with
                       a null. */
}

```

```

    return(ascii);
}

/*****
/* Convert a byte of binary coded decimal data to integer format. */
/* No check is made to ensure input data really IS in BCD format. */
int bcd_int(char bcd) /* Tested March 16, 1987. */
/* "bcd" is the BCD character to be converted. */
{
    int bcdint, result;
    /* Take the units by masking off the tens. */
    /* Then throw away the units and keep
       the tens.*/
    bcdint = 0x00ff & (int) bcd;
    result = 0x000f & bcdint;
    /*Multiply the tens by 10, and add to result.*/
    result += 10 * (bcdint >> 4);
    return(result);
}

/*****
/* Convert a character to hexadecimal ASCII string format. */
char *ctoh(char byte)
{
    static char ascii[HSTRLLEN];
    int byteint, nibble, base;

    byteint = 0x00ff & ((int) byte); /* Convert to integer. */
    nibble = byteint >> 4; /* Get the tens digit. */
    /* Find out whether the nibble is in the range [0-9], in which
       case its ASCII representation starts at 0x30 (48 decimal), or
       [10-15], in which case the ASCII representation starts at
       A = 0x41 (65 decimal). In the latter case, add the value of the
       nibble to 05-10 = 55. */
    base = (nibble >= 10) ? 55 : 48;
    ascii[0] = base + nibble;
    nibble = byteint & 0x0f; /* Get the units digit. */
    base = (nibble >= 10) ? 55 : 48;
    ascii[1] = base + nibble;
    ascii[2] = NULL; /* Terminate the string with
                       a null. */

    return(ascii);
}

/*****
/* This routine converts an integer to a binary coded decimal character.
   Since 99 is the largest legitimate BCD number, the argument "decimal"
   is taken modulo 100. */
char int_bcd(int decimal)
/* "decimal" is the number to be converted. */
{
    int result;

    /* Make sure decimal is a positive number. */

    decimal = (decimal < 0) ? -decimal : decimal;
    decimal %= 100; /* If decimal is too big, take

```

```

        it modulo 100. */
result = (decimal / 10) << 4;    /* Get the tens and shift them into the
        high order half of the byte. */
result += decimal % 10;        /* Add in the units. */
return((char) result);
}

/*****
/* itoa - convert n to characters in s.
This program is from TOOLHORKS C/80, Version 3.1, by Walt Bilofsky. */
char *itoa(int n, char s[])
{
    static int c, k;
    static char *p, *q;

    if ((k = n) < 0)
        k = -k;
    q = p = s;
    do {
        *p++ = k % 10 + '0';
    } while (k /= 10);
    if (n < 0) *p++ = '-';
    *p = 0;
    while (q < --p) {
        c = *q; *q++ = *p; *p = c; }
    return (s);
}

/*****
/* tolower - if the input is in [A..Z], convert to lower case
This program is from TOOLHORKS C/80, Version 3.1, by Walt Bilofsky. */
char tolower(int c)
{
    if ('A' <= c && c <= 'Z')
        return (c + 0x20);
    return c;
}

/*****
/* Convert an unsigned integer to hexadecimal ASCII string format. */
char *uitoh(unsigned int word)
{
    static char ascii[HEXINTSTRLEN + 1];
    unsigned int nibble;
    int i;

    ascii[HEXINTSTRLEN] = NULL;
    for (i=0; i < HEXINTSTRLEN; ++i) {
        /* Get the current nibble, in order from most to least significant. */
        nibble = 0x000f & (word >> (4 * (3 - i)));
        /* If nibble >= 10, convert it to a letter from 'A' to 'F'.
           If nibble < 10, convert it to a letter from '0' to '9'. */
        ascii[i] = (nibble >= 10) ? ('A' + nibble - 10) : ('0' + nibble);
    }
    return(ascii);
}

```

M. FILENAME DELAY.H

```
/* This file contains external prototyping declarations for all functions
in "delay.s". */
```

```
extern void delay(int n);
```

N. FILENAME DELAY.S

```

; delay.s
; Adapted from a program by Mr. David Rigmaiden of the
; Space Systems Academic Group at the Naval Postgraduate School.

#define LOOPCOUNT 1041

; Delay for n hundredths of a second.
; void delay(n)
; int n; /* The number of hundredths of seconds of delay desired. */

export delay
region code

delay: push ix ; t=15T.
; Cause ix to point to the first parameter.
ld ix,4 ; t=14T.
add ix,sp ; t=15T.
ld c,(ix+0) ; t=19T.
ld b,(ix+1) ; t=19T.
LOOP1: ld de,$LOOPCOUNT ; t=10T.
LOOP2: dec de ; t= 6T. Count down to zero in LOOP2.
ld a,d ; t= 4T.
or e ; t= 4T.
jp nz,LOOP2 ; t=10T. Inner loop t=24T.
dec bc ; t= 6T. Repeat LOOP1 until time is up.
ld a,b ; t= 4T.
or c ; t= 4T.
jp nz,LOOP1 ; t=10T. Outer loop t=(34+24*LOOPCOUNT)T.
pop ix ; t=14T. Restore ix to its initial value.
ret ; t=10T.
; Total Delay =(106+(34+24*LOOPCOUNT)*n)T.

; Solve n*10 ms = (106+(34+24*LOOPCOUNT)*n)T with T = 1/f = 400 ns to
; get n = LOOPCOUNT. f = 2.5 MHz. For n=100, LOOPCOUNT = 1041, leading
; to a delay of 1.0008 s for an error of 0.08%. For n=1,
; this leads to a delay of 10.05 ms instead of the 100 ms required, for
; and error of 0.5%.
```

O. FILENAME EXPMNT.H

```
extern char ad_read(char);
extern int adtoint(char addata,unsigned long multiplier);
extern void alter_page0(struct page0data * pagezero);
extern char bad_idea_to_record(char show);
extern char baro_switch(void);
extern char checkprt(void);
extern char colder_than(int reference);
extern void display_data_page(struct full_log_page * datapage);
extern void display_page0(struct page0data * pagezero);
extern void do_sweep(void);
extern void expmnt(void);
extern void initialize(void);
extern char listen(void);
extern char logevent(char event);
extern void log_menu(void);
extern void read_ad(void);
extern void shut_down(void);
extern void shut_down_no_log(void);
extern char ssdrmode(char mode);
extern char ssdr_status(void);
extern char voltages_low(void);
extern char we_launched(void);
```

P. FILENAME EXPMNT.C

```
/* expmnt.c */
```

```
#include "vibro.h"
#include "clock.h"
#include "convert.h"
#include "inout.h"
#include "main.h"
#include "power.h"
#include "newio.h"
#include "bubble.h"
#include "global.h"
```

```
char ad_read(char port);
int adtoint(char addata,unsigned long multiplier);
void alter_page0(struct page0data * pagezero);
char bad_idea_to_record(char show);
char baro_switch(void);
char checkprt(void);
char colder_than(int reference);
void display_data_page(struct full_log_page * datapage);
void display_page0(struct page0data * pagezero);
void do_sweep(void);
void expmnt(void);
char initialize(void);
char listen(void);
char logevent(char event);
void log_menu(void);
void monitor_heaters(void);
```

```

void post_launch(void);
void record(void);
void shut_down(void);
void shut_down_no_log(void);
char ssdrmode(char mode);
char ssdr_status(void);
void short_experiment(void);
void show_event(char event);
char voltages_low(void);
char we_launched(void);

/*****
/* This routine gets data from the analog to digital converter. */
char ad_read(char port)
{
    output(port,0);          /* You must write to the port before you
                             can read it. */
    delay(1);
    return(input(port));
}

/*****
/* This routine converts a byte of data from the A/D converter into an
integer. In order to reduce the amount of code generated by the compiler,
it uses no floating point operations.

    The routine assumes that the converted value lies on a line which passes
through the origin and whose slope (in some arbitrary units) is given by
the multiplier. Consequently, this routine always converts value of zero
to zero.

    To obtain the correct multiplier amounts to calculating the slope and
scaling it to permit integer operations to succeed.

    For example, assume that a value of 255 in the A/D converter (the
maximum possible) represents 15V. A difference of 1 in the value
read by the A/D converter represents
    15V / 255 divisions = 58.8235 mV/division.
Multiply this by 1E6 and round off to get the basic multiplier:
    58.8235 * 1E6 = 58824.

Using this multiplier will give results in units of volts. To get units
of tenths of volts, say, increase the multiplier by a factor of 10 to
588,240. The result will be an integer representing the chosen units;
the decimal point is implied to be to the left of the rightmost digit.

    To avoid an overflow upon multiplication, the multiplier should
be kept less than
    (2**32)/255 = 16,843,009.

The greatest achievable accuracy is obtained when the multiplier is scaled
up by multiples of 10 as much as possible without exceeding this limit.
*/
int adtoint(char addata,unsigned long multiplier)
{
    /* During compilation, this line will be flagged because it presents
    the possibility of truncation. The problem is not serious as
    long as the limit on the multiplier is observed, as discussed above. */
    unsigned long value; /* A long integer version of "addata". */

    value = (unsigned long) addata;

```

```

    return((int) (((value * multiplier) + 500000L) / 1000000L));
}

/*****
/* This routine allows the user to alter the flags and pointers in page zero
for the purpose of permitting program functions to be tested thoroughly.
Use caution in altering them. */

void alter_page0(struct page0data * pagezero)
{
    char data;                /* Holds a character from the keyboard. */
    char changes = FALSE;     /* TRUE if the page zero needs to be altered,
                               FALSE otherwise. We know that no unsaved
                               changes have been made to page 0 before this
                               routine is invoked, so we set this to FALSE
                               initially. */

    /* Variable "flag" is used to permit the values 0 and 1 to be displayed
       as FALSE and TRUE respectively. */
    static char *flag[] = {
        "FALSE",
        "TRUE"
    };

    /* Display this menu repetitively until choice Z is made. */
    while(TRUE) {
        printf("
A Toggle 'sweepstarted' flag from %s to %s. n r
B Toggle 'launchdone'   flag from %s to %s. n r
C Alter value of next available page from 0x%x = %d. n r
D Alter value of next available half page from %u to %u. n r
E Toggle 'full_experiment' flag from %s to %s. n r
F Specify the 'RECORD_start_time' (make this at least 12 hours before the n r
present to permit RECORD mode to be initiated.) n r
Z Exit this menu. n r",
            flag[pagezero->sweepstarted ? 1 : 0 ],
            flag[pagezero->sweepstarted ? 0 : 1 ],
            flag[pagezero->launchdone ? 1 : 0 ],
            flag[pagezero->launchdone ? 0 : 1 ],
            pagezero->page, pagezero->page,
            pagezero->halfpage,
            (pagezero->halfpage == 0) ? 1 : 0,
            flag[pagezero->full_experiment ? 1 : 0],
            flag[pagezero->full_experiment ? 0 : 1]
        );

        /* Input a character, convert it to lower case, and display it. */
        data = tolower(termin());
        printf("%c n r", data);
        switch (data) {
            case 'a': /* Complement the "sweepstarted" flag. */
                pagezero->sweepstarted = !pagezero->sweepstarted;
                changes = TRUE;
                break; /* Complement the "launchdone flag. */
            case 'b':
                pagezero->launchdone = !pagezero->launchdone;

```

```

        changes = TRUE;
        break;
    case 'c': /* Ask the user for a page number. Let this be the
               next page used for recording items in the log. */
        pagezero->page = getpageno();
        changes = TRUE;
        break;
    case 'd': /* Complement the "halfpage" number. */
        pagezero->halfpage = (pagezero->halfpage == 0) ? 1 : 0;
        changes = TRUE;
        break;
    case 'e': /* Complement the "full_experiment" flag. */
        pagezero->full_experiment = !pagezero->full_experiment;
        changes = TRUE;
        break;
    case 'f': /* Ask the user for a new "RECORD_start_time". */
        get_time(&(pagezero->RECORD_start_time));
        changes = TRUE;
        break;
    case 'z': /* If any changes have been made, store them in page
               0 and quit this routine. */
        if (changes) {
            if (!bubio(BWRITE,0,(char *) page0_buffer)) {
                printf("Update to page 0 failed.\n");
            }
        }
        return;
    default:
        printf("Use a valid letter, please.\n");
}
}
}

```

```

/*****

```

```

/* This routine checks to see when RECORD mode was last initiated.
   If this time was within the last 12 hours, the routine returns TRUE,
   meaning that it is not a good idea to enter RECORD mode now. This
   will avoid a situation where RECORD mode is restarted in the middle
   of a mission, wiping out the recorded data. "show" must be TRUE to display
   the time when RECORD mode can begin. If it is FALSE, the display is
   suppressed. */

```

```

char bad_idea_to_record(char show)
{
    struct datetime    current_time;
    struct idatetime icurrent_time; /* Integer version of
                                     current time. */
    struct idatetime istored_time; /* Integer version of
                                     stored time. */
    struct idatetime iRECORD_delay_time; /* Integer format of time when
                                           RECORD mode can begin. */
    struct idatetime iRECORD_delay_constant; /* Integer format of minimum
                                                time between successive
                                                startings of RECORD mode. */

    iRECORD_delay_constant.imonth = iRECORD_delay_constant.idate =
        iRECORD_delay_constant.iminute = iRECORD_delay_constant.isecond = 0;
}

```

```

iRECORD_delay_constant.ihour = RECORD_DELAY;

/* Get the current date and time and convert to integer format. */
clockread(&current_time);
clockint(&current_time,&icurrent_time);

/* Get the date and time stored in the bubble memory as the last time
   that RECORD mode was initiated and convert to integer format. */
clockint(&(pagezero->RECORD_start_time),&istored_time);

/* Add the two dates and times to get the next time when RECORD mode can
   be initiated. */
clocksum(&iRECORD_delay_time,&istored_time,&iRECORD_delay_constant);

if (show) {
    printf("Current time: ");
    dump_clock(&current_time);
    printf("Time when RECORD mode last was begun: ");
    dump_clock(&(pagezero->RECORD_start_time));
    printf("Time when RECORD mode can be begun again: ");
    dump_iclock(&iRECORD_delay_time);
}

/* Return TRUE if the current date and time is less than RECORD_DELAY
   hours after the stored date and time. Otherwise, return FALSE. */
return(clockcompare(&iRECORD_delay_time,&icurrent_time));
}

/*****
/* Check to see if the barometric pressure switch tripped.
   Make an entry in the log and return TRUE if so; return FALSE otherwise. */
*****/
char baro_switch(void)
{
    char addata;                /* Holds a character from port READC1. */

    /* If the BARO_ON bit of the READC1 port is TRUE, then the barometric
       switch has been triggered. */
    addata = input(READC1);
    if (addata & BARO_ON) {
        printf("Barometric switch triggered.n\r");
        logevent(DPRESSURE);
    }
}

/*****
/* This routine checks to see if there is a printer connected to the
   controller. It returns TRUE if there is one, FALSE otherwise. */
*****/
char checkprt(void)
{
    /* If the TERMON bit of the READC1 port is 0, then a terminal
       is connected. In this case return TRUE; FALSE otherwise. */

    return(( input(READC1)) & TERMON);
}

```

```

/*****/
/* This function displays the data in page zero. */
void display_page0(struct page0data * pagezero)
{
    printf("Sweepstarted = ");
    if (pagezero->sweepstarted)
        printf("TRUE ");
    else
        printf("FALSE ");
    printf("Launchdone = ");
    if (pagezero->launchdone)
        printf("TRUE ");
    else
        printf("FALSE ");
    printf("Full-expmnt = ");
    if (pagezero->full_experiment)
        printf("TRUE n r");
    else
        printf("FALSE n r");
    printf("Last time RECORD mode was initiated: ");
    dump_clock( &pagezero->RECORD_start_time );

    printf("n rNext page = 0x%x = %u ",    pagezero->page,pagezero->page);
    printf("Next halfpage = 0x%x = %u n r",    pagezero->halfpage,
        pagezero->halfpage);
}

/*****/
/* This function returns TRUE if the bubble memory's temperature is below
the reference value; FALSE otherwise. */
char colder_than(int reference)
{
    char addata;          /* Holds a character from the A/D. */
    int  temperature;     /* The current temperature in degrees K. */

    /* Read in the temperature of the bubble memory. */
    addata = ad_read(TEMP4);
    temperature = adtoint(addata,MULT_TEMP);
    return((temperature < reference) ? TRUE : FALSE);
}

/*****/
/* This function displays a page of data. */
void display_data_page(struct full_log_page *datapage)
{
    char addata;          /* Holds a character of data from the A/D. */
    int  page;            /* The desired page number. */
    int  halfpage;        /* The current halfpage number. */
    int  i;               /* Counts through the valid A/D addresses. */
    int  value;           /* The data from the A/D converted into useful
                           units. */

    printf("Which page of data do you want to see? n r");
    page = getpageno();
    if (!bubio(BREAD,page,log_buffer)) {
        printf("Couldn't read page %u. n r",page);
    }
}

```

```

    return;
}
printf("Contents of page 0x%x = %d: n r",page,page);
for (halfpage=0;halfpage < BLOCKS_PER_PAGE;halfpage) {
    printf("Half page %d: n r t",halfpage);
    dump_clock( &(datapage->half_page[halfpage].clock) );
    show_event( datapage->half_page[halfpage].event );

    /* "adcaption" is defined in file "global.c". */
    for (i=0;i < ADPOINTS;++i) {
        addata = datapage->half_page[halfpage].atod[i];
        printf("%-24s=%3.0d=",adcaption[i],addata);
        if (i <= 2) { /* The A/D reading is a voltage, in this case. */
            value = adtoint(addata,(i==2)?MULT_10V : MULT_20V);
            printf("%c%2.0d.%02.0dV ",(i==1)?'-':'+',
                value/100,value%100);
        } else { /* The A/D reading is a temperature, in this case. */
            value = adtoint(addata,MULT_TEMP);
            printf("%6.0dK ",value);
        }
        /* Print two points per line. */
        if ((i != i % 2) || i == ADPOINTS - 1)
            printf(" n r");
    }
}

/*****
/* This function causes the "sweep" to be performed. */
void do_sweep(void)
{
    printf("Turn on SS DR and A/D Converter and place SS DR in SWEEP mode. n r");
    logevent(power_write(ADON) ? CSONAD : CFONAD);
    logevent(power_write(SSDRON) ? CSONSSDR : CFONSSDR);
    logevent(ssdrmode(SWEEP) ? CSSWEEP : CFSWEEP);
    printf("Wait 10 seconds before starting sweep. n r");
    timeout(10,SECONDS);
    /* Wait for timeout or for a key to be pressed. */
    while(!timeout(NULL,NULL)) {
        if (look_ahead_discard())
            break;
    }
    printf("Turn on VCO. Wait 13 minutes. n r");
    logevent(power_write(VCOON) ? CSONVCO : CFONVCO);
    timeout(13,MINUTES);
    while (TRUE) {
        if (ssdr_status() == OPCOMP) {
            logevent(DOPCOMP);
            break;
        }
        if (timeout(NULL,NULL) || look_ahead_discard()) {
            logevent(DNOOPCOMP);
            break;
        }
    }
    printf("Sweep phase is over. Turn off VCO, A/D Converter, and SS DR. n r");
}

```

```

logevent(power_write(VCOFF) ? CSOFFVCO : CPOFFVCO);
logevent(power_write(SSDROFF) ? CSOFFSSDR : CPOFFSSDR);
logevent(power_write(ADOFF) ? CSOFFAD : CPOFFAD);
logevent(DSHEEP);
}

/*****
/* This function performs the Vibro-acoustic experiment. */
void expmnt(void)
{
    char mission_status; /* Can be DAPUON, DLAUNCH, DOPCOMP, DNOOPCOMP,
                           or DABORT. Used to control program flow. */

    if (!initialize()) {
        printf("The bubble memory log is full. \n\n");
        Running the experiment anyway. n r");
    }

    /* Check to see whether we should operate the full experiment or not. */
    if (!pagezero->full_experiment) {
        short_experiment();
        return;
    }

    /* Check the sweepstarted flag in page zero of the controller's bubble
       memory. It's TRUE if the sweep has been started previously,
       false otherwise. */
    if (!pagezero->sweepstarted) {
        printf("Starting the sweep. n r");
        do_sweep();
    } else {
        printf("Sweep was done previously, so we're skipping it. n r");
    }

    /* Check the launchdone flag in page zero of the controller's bubble
       memory. It's TRUE if the launch has already taken place; FALSE
       otherwise. */
    if (!pagezero->launchdone) {
        /* Keep on listening, until you detect either the APU, or the
           launch. If you run out of time, assume the mission
           was aborted. */
        printf("We haven't launched yet. Listening for the APU. n r");
        mission_status = listen();
        printf("Turning on the SSDR, because listen() detected something. n r");
        logevent(power_write(ADON) ? CSONAD : CFONAD);
        logevent(power_write(SSDRON) ? CSOSSDR : CFONSSDR);
        if (mission_status == DAPUON) {
            printf("APU is on. Initiate a 10 minute timeout. n r.
Placing SSDR in SCROLL mode. n r");
            logevent(ssdrmode(SCROLL) ? CSSROLL : CFSCROLL);
            timeout(10, MINUTES);
            while (TRUE) {
                if (we_launched() == DLAUNCH) {
                    printf("We launched. n r");
                    mission_status = DLAUNCH;
                    break;

```

```

    }
    if (timeout(NULL,NULL) || look_ahead_discard()) {
        printf("We timed out and are aborting the mission.n r");
        mission_status = DABORT;
        logevent(DABORT);
        break;
    }
}
}
} else { /* Launch was done previously. */
    logevent(PRIORLAUNCH);
    mission_status = PRIORLAUNCH;
    printf("We have previously launched and are in space now.n r");
    Assume mission has been successfully completed.n r");
}
if (mission_status == DLAUNCH) {
    printf("Putting the SSDR in LAUNCH mode.n r");
    logevent(ssdrmode(LAUNCH) ? CSLAUNCH : CFLAUNCH);
    timeout(3,MINUTES);
    while (TRUE) {
        /* If we haven't recorded a completed launch, check the barometric
        switch. If it has been triggered, then we should record one. */
        if (!pagezero->launchdone)
            baro_switch();
        if (ssdr_status() == DOPCOMP) {
            printf("SSDR reported OP COMPLETE.n r");
            logevent(DOPCOMP);
            break;
        }
        if (timeout(NULL,NULL) || look_ahead_discard()) {
            printf("SSDR never reported OP COMPLETE. We timed out.n r");
            logevent(DNOOPCOMP);
            break;
        }
    }
}
if (mission_status != DABORT)
    post_launch();
}

/*****
/* This routine reads page 0 from bubble memory in order to initiate the
experiment properly. */
char initialize(void)
{
    int i;          /* A counter which permits more than one attempt
                     to read the bubble memory. */
    char power_port; /* Holds the status of the power subsystem. */

    printf("Read from page 0 of the bubble memory.n r");
    /* Attempt to read from page 0 up to BTRIES times before giving up. */
    for(i=0; !bubio(BREAD,0,page0_buffer) && i <= BTRIES; ++i);
    display_page0(pagezero);
    if (pagezero->page > MAXPAGE) {
        return(FALSE);
    }
}

```

```

}

for(i=0;i < BTRIES;++i) {
    if(bubio(BREAD,pagezero->page,log_buffer)) {
        printf("logevent(INITIALIZE) n r");
        logevent(INITIALIZE);
        power_port=power_status();
        if (VCOFF & power_port) {
            printf("Turning the VCO power subsystem off. n r");
            logevent(power_write(VCOFF) ? CSOFFVCO : CPOFFVCO);
        }
        if (HEATOFF & power_port) {
            logevent(power_write(HEATOFF) ? CSOFFHEAT : CPOFFHEAT);
            printf("Turning the heater subsystem off. n r");
        }
        return(TRUE);
    }
}
return(FALSE);
}

/*****
/* This function returns DAPUON if the APU is on; DLAUNCH if the shuttle
   has launched; FALSE if neither event is detected, but exit is forced by
   the pressing of any key on the terminal. */
char listen(void)
{
    char portcl;          /* This holds the contents of NSC810 #1 port c.*/

    printf("Turning Matched Filter on.  Wait for detection or a keystroke. n r");
    /* Turn on the matched filter, and listen for the APU.  If the matched
       filter is already on, this command has no effect. */
    logevent(power_write(MATFON) ? CSONMATF : CFONMATF);
    while (TRUE) {

        if (we_launched() == DLAUNCH)
            return(DLAUNCH);

        portcl = input(READCL);
        if (portcl & APU_ON) {
            printf("APU detection occurred. n r");
            logevent(DAPUON);
            return(DAPUON);
        }

        /* Exit this function if any key on the terminal is pressed. */
        if (look_ahead_discard())
            return(DUSERNOAPU);
    }
}

/*****
/* Log an event. This function returns TRUE if the event was logged

```

```

    satisfactorily, FALSE otherwise. */
char logevent(char event)
{
    int i;                /* A counter. */
    char *buffer_ptr;      /* A pointer into the log page buffer. */
    char full;             /* TRUE when all available pages are
                           used up. */
    buffer_ptr = log_buffer; /* Make buffer_ptr point to the start of
                           the log buffer. */
    full = pagezero->page > MAXPAGE;

    /* If the bubble memory is full, there's no point in going on. Return. */
    if (full)
        return(FALSE);

    /* Blank out the buffer if this is a new page. This guarantees that
    old data won't reside in the upper half-page when the new page is
    written to the bubble memory. */
    if (pagezero->halfpage == 0) {
        for (i=0; i < PAGELENGTH; ++i) {
            (*buffer_ptr++) = 0x00;
        }
    }

    /* Fill the current log block with new data to be logged. */
    clockread(
        &{
            log_page->
                half_page[pagezero->halfpage].clock
        }
    );
    log_page->
        half_page[pagezero->halfpage].event = event;

    /* Read the A/Ds and put their contents in the log, too. */
    for (i=0; i < ADPOINTS; i++) {
        log_page->
            half_page[pagezero->halfpage].atod[i]
                = ad_read(adport[i]);
    }

    if (event == CSSHEEP || event == CFSHEEP) {
        pagezero->sweepstarted = TRUE;
    }
    if (event == DPRESSURE) {
        pagezero->launchdone = TRUE;
    }

    /* Write the new page of data to the bubble memory. */
    bubio(BWRITE, pagezero->page, log_buffer);
    if (pagezero->halfpage >= BLOCKS_PER_PAGE - 1) {
        if (++(pagezero->page) > MAXPAGE) {
            printf("Bubble memory is all used up.\n");
            return(FALSE);
        }
        pagezero->halfpage = 0;
    } else {

```

```

        ++(pagezero->halfpage);
    }

    /* Update page 0 in the bubble memory. */
    bubio(BHRITE,0,page0_buffer);
    return(TRUE);    /* If you got this far, you know you haven't yet
                      run out of bubble memory. Return TRUE to show
                      successful logging of an event. */
}

/*****
/* This routine provides a menu of choices for examining or changing the
contents of the bubble memory. */
void log_menu(void)
{
    char data;    /* Holds a character from the keyboard. */

    /* Read page 0 from the bubble memory. */
    if (!bubio(BREAD,0,page0_buffer)) {
        printf("Couldn't read page 0. n r");
        return;
    }

    /* Display the menu repetitively until Z is chosen. */
    while (TRUE) {
        printf("
A Display page 0. n r
B Display a page of the log. n r
C Alter the contents of page 0. n r
Z Exit this menu. n r"
);
        data = tolower(termin());
        printf("%c n r",data);
        switch (data) {
            case 'a':
                display_page0(pagezero);
                break;
            case 'b':
                display_data_page(log_page);
                break;
            case 'c':
                alter_page0(pagezero);
                break;
            case 'z':
                return;
            default:
                printf("Use a valid letter please. n r");
        }
    }
}

/*****
/* This routine monitors the temperature of the bubble memory used for
logging data and operates the heaters to keep the temperature within

```

```

    the desired range. */
void monitor_heaters(void)
{
    char power_port;    /* Holds the status of the power subsystem. */

    power_port = power_status();

    /* If it is cold enough, and if the heater is not yet on, turn it on. */

    if (colder_than(MIN_DESIRABLE_TEMP) && (HEATOFF & power_port)) {
        printf("Turn on the heaters. n r");
        logevent(power_write(HEATON) ? CSONHEAT : CFONHEAT);
        return;
    }

    /* It is is warm enough, and if the heater is already on, turn it off. */

    if (!colder_than(MAX_DESIRABLE_TEMP) && (HEATOFF & power_port) ) {
        printf("Turn the heaters off. n r");
        logevent(power_write(HEATOFF) ? CSOFFHEAT : CFFOFFHEAT);
    }
}

/*****
void post_launch(void)
{
    printf("We're shutting down all power. n r");
    shut_down();
    while (TRUE) {
        printf("Reading A/Ds every 5 minutes. n r");
        timeout(5,MINUTES);
        while (!timeout(NULL,NULL)) {
            monitor_heaters();
            if (!pagezero->launchdone)
                baro_switch();
            if (look_ahead_discard())
                break;
        }
        logevent(READAD);
        if (voltages_low()) {
            printf("Voltages are too low. Terminate the experiment. n r");
            logevent(TERMINATE);
            break;
        }
    }
}

*****/

/* This routine performs the RECORD phase of the abridged experiment. */
void record(void)
{
    printf("Entering RECORD mode. n r");
    Turning on SSDR and A/D Converter. n r");
    /* Store current time in page 0. This is a record of the time when
    RECORD mode last was begun. The next time logevent() is called,
    the data will actually be stored in page 0. */

```

```

clockread(&(pagezero->RECORD_start_time));
logevent(power_write(ADON) ? CSONAD : CFONAD);
logevent(power_write(SSDRON) ? CSOSSDR : CFONSSDR);
logevent(ssdrmode(RECORD) ? CSRECORD : CFRECORD);
printf("Initiating a 20 minute timeout. n r");
timeout(20,MINUTES);
while (TRUE) {
    /* If we haven't yet launched, check to see if the barometric
       switches have fired or not. */
    if (!pagezero->launchdone) {
        baro_switch();
    }
    if (ssdr_status() == OPCOMP) {
        logevent(DOPCOMP);
        break;
    }
    if (timeout(NULL,NULL) || look_ahead_discard()) {
        logevent(DNOOPCOMP);
        break;
    }
}
printf("Record phase is over. Turn off A/D Converter and SSDR. n r");
logevent(power_write(SSDROFF) ? CSOFFSSDR : CFFOFFSSDR);
logevent(power_write(ADOFF) ? CSOFFAD : CFFOFFAD);
}

/*****
/* This routine operates an abbreviated version of the experiment which
   avoids doing the "sweep", and uses only RECORD mode in the SSDR. */
void short_experiment(void)
{
    char showflag;          /* This flag is TRUE to make bad_idea_to_record()
                           display computed times; FALSE otherwise. */
    if (pagezero->launchdone) {
        printf("We have previously launched and are in space now. n r");
        logevent(PRIORLAUNCH);
    }

    while(!pagezero->launchdone) {
        /* If RECORD mode was initiated too recently, we don't
           want to try it again. Wait for a suitable interval to elapse before
           continuing. bad_idea_to_record() knows how long this is.
           Alternatively, the user can press a key to avoid waiting. */

        showflag = TRUE; /* Have bad_idea_to_record() display computed
                           times the first time through. */
        while (bad_idea_to_record(showflag)) {
            showflag = FALSE;
            if (look_ahead_discard())
                break;
        }
        /* Wait for indications of a launch. */
        listen();
        /* Enter RECORD mode. */
        record();
    }
}

```

```

        if (!pagezero->launchdone)
            baro_switch();
    }
    /* Now that we're in space, perform post-launch operations. */
    post_launch();
}

/*****
/* This function displays the meaning of an event code. */
void show_event(char event)
{
    /* "event" is an index into the following array of messages.
    It is one of the event codes given in the "vibro.h" file.
    If someone changes it, someone had better change these messages to
    correspond, or the results will be disappointing.*/
    static char *message[] = {
        "Initialization. n r",
        "Sweep-mode command issued. n r",
        "Sweep-mode command accepted. n r",
        "Sweep-mode command not accepted. n r",
        "Sweep-mode completion detected. n r",
        "APU detected. n r",
        "Scroll-mode command issued. n r",
        "Scroll-mode command accepted. n r",
        "Scroll-mode command not accepted. n r",
        "Launch detected. n r",
        "Launch-mode command issued. n r",
        "Launch-mode command accepted n r",
        "Launch-mode command not accepted. n r",
        "Barometric switch detection. n r",
        "SSDR did not give OP COMPLETE within the allotted time. n r",
        "SSDR reported OP COMPLETE. n r",
        "Mission abort inferred. n r",
        "SSDR ON command issued. n r",
        "SSDR ON command succeeded. n r",
        "SSDR ON command failed. n r",
        "SSDR OFF command issued. n r",
        "SSDR OFF command succeeded. n r",
        "SSDR OFF command failed. n r",
        "VCO OFF command issued. n r",
        "VCO OFF command succeeded. n r",
        "VCO OFF command failed. n r",
        "VCO ON command issued. n r",
        "VCO ON command succeeded. n r",
        "VCO ON command failed. n r",
        "A/D OFF command issued. n r",
        "A/D OFF command succeeded. n r",
        "A/D OFF command failed. n r",
        "A/D ON command issued. n r",
        "A/D ON command succeeded. n r",
        "A/D ON command failed. n r",
        "MATCHED FILTER OFF command issued. n r",
        "MATCHED FILTER OFF command succeeded. n r",
        "MATCHED FILTER OFF command failed. n r",

```

```

    "MATCHED FILTER ON command issued.n r",
    "MATCHED FILTER ON command succeeded.n r",
    "MATCHED FILTER ON command failed.n r",
    "BUBBLE MEMORY HEATER OFF command issued.n r",
    "BUBBLE MEMORY HEATER OFF command succeeded.n r",
    "BUBBLE MEMORY HEATER OFF command failed.n r",
    "BUBBLE MEMORY HEATER ON command issued.n r",
    "BUBBLE MEMORY HEATER ON command succeeded.n r",
    "BUBBLE MEMORY HEATER ON command failed.n r",
    "READ A/D command issued.n r",
    "Experiment terminated.n r",
    "User interrupted the wait for APU detection.n r",
    "Invalid command.n r",
    "Launch occurred before the last program initialization.n r",
    "RECORD mode command to SDR succeeded.n r",
    "RECORD mode command to SDR failed.n r."
};
printf(message[event]);
}

/*****
/* This routine removes power from any relays which have it, and logs the
fact. */
void shut_down(void)
{
    char power_port;    /* Holds the status of the power subsystem. */
    power_port = power_status();
    /* Remove power from all subsystems which currently have power. */
    if (SSDROFF & power_port)
        logevent(power_write(SSDROFF) ? CSOFFSSDR : CPOFFSSDR);
    if (VCOOFF & power_port)
        logevent(power_write(VCOOFF) ? CSOFFVCO : CPOFFVCO);
    if (ADOFF & power_port)
        logevent(power_write(ADOFF) ? CSOFFAD : CPOFFAD);
    if (MATFOFF & power_port)
        logevent(power_write(MATFOFF) ? CSOFFMATF : CPOFFMATF);
    if (HEATOFF & power_port)
        logevent(power_write(HEATOFF) ? CSOFFHEAT : CPOFFHEAT);
}

/*****
/* This routine removes power from any relays which have it. It does not
log the fact. */
void shut_down_no_log(void)
{
    char power_port;    /* Holds the status of the power subsystem. */
    power_port = power_status();
    /* Remove power from all subsystems. */
    if (SSDROFF & power_port)
        power_write(SSDROFF);
    if (VCOOFF & power_port)
        power_write(VCOOFF);
    if (ADOFF & power_port)
        power_write(ADOFF);
    if (MATFOFF & power_port)
        power_write(MATFOFF);
}

```

```

    if (HEATOFF & power_port)
        power_write(HEATOFF);
}

/*****
/* This routine sets the SSDR's mode. */
/* "mode" is a coded SSDR mode. See file "vibro.h". */

char ssdrmode(char mode)
{
    int i;           /* A counter. */
    char status;     /* Hexadecimal status, used for debugging. */

    /* Repeat the following code several times if the SSDR does not
       immediately appear to be successful. */
    for(i=0; i < TRIES; ++i) {
        output(SSDROUT, mode);    /* Output a mode command to the SSDR. */

        /* Wait for the SSDR to respond. */
        delay(2);    /* Delay 2 x 10 ms to get a valid
                       status. */

        status = ssdr_status();
        if (status == NORMOP) {
            /* Adding 1 to a mode gives the code for a successful operation. */
            printf("SSDR returned NORMOP in response to command 0x%x.n r",
                   mode);
            return(TRUE);
        }
        /* The SSDR did not confirm the proper mode was set.
           Try again. After you give up in disgust, signal
           failure by returning FALSE. */
    }
    printf("SSDR did not return NORMOP in response to command 0x%x.n r", mode);
    return(FALSE);
}

/*****
/* This routine gets the SSDR's status. */
char ssdr_status(void)
{
    return(input(SSDRIN));
}

/*****
/* This function returns TRUE if the power supply voltages are too low;
   FALSE, otherwise. */
char voltages_low(void)
{
    int voltage;      /* Holds the voltage in hundredths of a volt. */
    char addata;      /* Holds the voltage as read by the A/D. */

    /* Read in the voltage on the 10V bus. */
    addata = ad_read(VOLT2);
    /* Convert to hundredths of a volt. */

```

```

    voltage = adtoint(adata,MULT_10V);
    if (voltage < MIN_VOLTAGE_10) {
        return(TRUE);
    }
    return(FALSE);
}

/*****

char we_launched(void)
{
    char portdata;          /* Holds the port data. */

    /* Check to see if the barometric pressure switch tripped. */
    baro_switch();

    portdata = input(READC1);
    if ((VIB_ON | BARO_ON) & portdata) {
        printf("Launch detected. Turning matched filter off. n\r");
        logevent(DLAUNCH);
        logevent(power_write(MATOFF) ? CSOFFMATF : CPOFFMATF);
        return (DLAUNCH);
    }
    return(FALSE);
}

```

Q. FILENAME FPUTC.C

```

/* fputc.c */

#include "inout.h"
#include "vibro.h"
#include "expmnt.h"
#include "newio.h"

int fputc(int chr, void *device);

/* Bit 0 of the serial port is TRUE if the serial port is ready to write
   a character; FALSE otherwise.
   Bit 1 of the serial port is TRUE if the serial port is ready to read
   a character; FALSE otherwise. */
struct rs232c {
    unsigned int :6;
    unsigned int read_ready:1;
    unsigned int write_ready:1;
};

/* Implementation of fputc() as described in the Uniware Compiler manual.
   For the NSC800 controller, there is only one valid output device, namely
   the RS232C terminal. The variable "device" is therefore ignored.
   This module must be place in unilib libc.a using the uar.exe utility. */

int fputc(int chr, void *device)
{

```

```

struct rs232c portdata;
char port_status;

/* Allow the user to interrupt the display of data by use of control
   characters. */
allow_ctrl_interrupts();

/* The UNINARE manual specifies that this function must return -1
   if it cannot output a character. If there is no terminal attached,
   this is the case. */
if (!checkprt())
    return(-1);
do {
    /* Keep getting the status information for the RS232C data port
       until it is ready to accept data. */
    port_status=input(PRTCTRL);
    portdata = *(struct rs232c *) &port_status;
} while (!portdata.write_ready);
/* Otherwise, output the character and return it. */
output(PRTDATA,(char) chr);
return(chr);
}

```

R. FILENAME GLOBAL.H

```

/* This file contains external prototyping declarations of data used globally
   throughout the control program. */

```

```

extern char prtconnected;
extern char tempbuffer[PAGELNGTH];
extern struct datetime clock;
extern struct idatetime waketime;
extern struct power_port_fmt power_ports;
extern char adport[ADPOINTS];
extern char page0_buffer[PAGELNGTH];
extern char log_buffer[PAGELNGTH];
extern struct page0data *pagezero;
extern struct full_log_page *log_page;
extern char *adcaption[];

```

S. FILENAME GLOBAL.C

```

/* global.c */
/* This file contains the declarations of global variables needed by
   the control program. */

#include "vibro.h"

char prtconnected; /* TRUE if there is a terminal attached, FALSE
                   otherwise. */

char tempbuffer[PAGELNGTH]; /* A temporary buffer. */

struct datetime clock; /* The most recently read time will be

```

```

                                stored here. */

struct idatetime waketime;      /* The most recently read integer version of
                                time will be stored here. */

struct power_port_fmt power_ports;

/* This is a list of A/D channels in use, and what they're used for.
   Make sure ADPOINTS = the number of transducers in use. */
char adport[ADPOINTS] = {
    VOLT0, VOLT1, VOLT2,
    TEMP0, TEMP1, TEMP2, TEMP3, TEMP4, TEMP5, TEMP6
};

char page0_buffer[PAGELENGTH]; /* A buffer for bubble memory page 0. */
char log_buffer[PAGELENGTH]; /* A buffer for bubble memory log data. */

struct page0data *pagezero;     /* A pointer to the page0_buffer. */
struct full_log_page *log_page; /* A pointer to the log_buffer. */

/* The following captions should match the A/D port assignments,
   in order. See the vibro.h header file.*/
char *adcaption[] = {
    "+20V Bus",
    "-20V Bus",
    "+10V Bus",
    "T, shelf above BMC",
    "T, underside of speaker",
    "T, shelf above batteries",
    "T, batteries",
    "T, controller backplane",
    "T, card 8 of BMC",
    "T, card 9 of BMC"
};

```

T. FILENAME INITIAL.H

```

/* This file contains external prototyping declarations for all functions
in "initial.c". */

extern void inithardwarel(void);

```

U. FILENAME INITIAL.C

```

/* initial.c */

#include "vibro.h"
#include "inout.h"
#include "newio.h"

void inithardwarel(void);

```

```

/*****
/* This routine initializes the NSC810A ports. */
void inithardware(void)
{
    output(MDR1,0x00);    /* A 0x00 in the Mode Definition Register
                           of NSC810 #1 puts port A1 into basic I/O
                           mode. */

    output(DDRA1,0xff);   /* Set port A1 to output.
                           A1 is used to output command codes to
                           the SSDR. */

    output(DDRB1,0xff);   /* Set port B1 to output.
                           B1 is used to send command codes to the
                           power subsystem. */

    output(DDRC1,0x30);   /* Set port C1 to input/output.
                           Bits are defined in vibro.h */

    output(TM01,0x00);     /* Stop timer 0 of NSC810A #1. You must do this
                           before changing the timer's mode. */

    output(TM01,0x25);     /* Set timer mode to generate square waves
                           without pre-scaling, and with single-
                           precision selected, meaning only the low-
                           order byte is ever read. */

    output(TOLB1,0x07);    /* Set low-order and high-order byte for timer. */
    output(TOHB1,0x00);    /* The modulus is thus 7. After 2*(7+1)
                           cycles, the timer is reloaded. Since the
                           NSC800 clock has a frequency of 4.9152 Mhz,
                           and this is divided by 2, the timer produces
                           one pulse every 6.51 mus, for a 153.6 KHz
                           signal. This signal is fed to the UART where
                           it is divided by 16 to give a 9600 BAUD clock
                           for serial communications. */

    output(START01,0x00); /* Restart timer 0 of NSC810A #1 by writing
                           anything to it. */

    output(MDR2,0x00);     /* A 0x00 in the Mode Definition Register
                           of NSC810 #2 puts port A2 into basic I/O
                           mode. */

    output(DDRA2,0x00);    /* Set port A2 to input.
                           A2 is used to read status codes from the
                           SSDR. */

    output(DDRB2,0x00);    /* Set port B2 to input.
                           B2 is used to read relay position codes
                           from the power subsystem. */

    output(DDRC2,0x31);    /* Set port C2 to input/output.
                           Bits are defined in vibro.h */

    output(TM02,0x00);     /* Stop timer 0 of NSC810A #2. You must do this
                           before changing the timer's mode. */

    output(TM02,0x25);     /* Set timer mode to generate square waves
                           without pre-scaling, and with single-
                           precision selected, meaning only the low-
                           order byte is ever read. */

    output(TOLB2,0x01);    /* Set low-order and high-order byte for timer. */
    output(TOHB2,0x00);    /* The modulus is thus 1 decimal. After 2*(1+1)
                           cycles, the timer is reloaded. Since the
                           NSC800 clock has a frequency of 4.9152 Mhz,
                           and this is divided by 2, the timer produces
                           one pulse every 1.628 mus, for a 614.4 KHz
                           signal. This signal is fed to the A/D
                           converters. For a 640 KHz clock, the A/Ds

```

```

        will complete a conversion in about 100 mus.
        We are not far from 640 KHz, so should get
        comparable performance. */
output(START02,0x00); /* Restart timer 0 of NSC810A #2 by writing
                        anything to it. */
output(BCLRC2,0x30); /* Ensure that power is not applied to the
                        bubble memory and that a reset is applied to
                        it. This should be done when the NSC810 first
                        receives power, but we leave nothing for
                        granted. */
}

```

V. FILENAME INOUT.H

```

/* This file contains external prototyping declarations for all functions
in "inout.c". */

```

```

extern void allow_ctrl_interrupts(void);
extern void dump(unsigned int address, unsigned int length);
extern char gethex(void);
extern unsigned int gethexint(void);
extern int getint(void);
extern int getpageno(void);
extern char look_ahead(char *character);
extern char look_ahead_discard(void);
extern void portdump(char *string);
extern char termin(void);
extern void testinput(void);
extern void testoutput(void);

```

W. FILENAME INOUT.C

```

/* inout.c */

#include "vibro.h"
#include "convert.h"
#include "newio.h"
#include "bubble.h"
#include "expmnt.h"
#include "global.h"
#include "main.h"

void allow_ctrl_interrupts(void);
void dump(unsigned int address, unsigned int length);
char gethex(void);
unsigned int gethexint(void);
int getint(void);
int getpageno(void);
char look_ahead(char *character);
char look_ahead_discard(void);
char termin(void);
void testinput(void);
void testoutput(void);

```

```

/* console_buffer is shared by look_ahead() and termin(). If look_ahead()
reads a character in, it puts it in the buffer and sets the
console_data_available flag to true.
termin() will look first in the buffer for
input from the console. If it finds any, it will set console_data to false
and return the character in the buffer. Otherwise it will try to get a
character from the console in the usual way. */
static char console_buffer;
static char console_data_available = FALSE;

/*****
/* This routine processes the special characters CTRL S and CTRL Y from
the Keyboard. */
void allow_ctrl_interrupts(void)
{
    char conchar;          /* The character of console input data itself. */
    char char_available; /* TRUE if there is a character available for
                           input from the console; FALSE otherwise. */

    /* If there is a S character in the RS232C input port, then read it
    in using termin() and loop until another S is given. Thus, S
    serves as a toggle for stopping and starting output. */

    /* See if there is a character available,
    and if so, put it in conchar.*/
    char_available = look_ahead(&conchar);

    if (char_available) {
        switch(conchar) {
            case CTRLS:
            case CTRLY:
                /* Call termin() to empty the buffer and handle the
                control character. */
                termin();
                break;
            default:
                break;
        }
    }
}

/*****
/* This routine produces a hexadecimal dump of any section of memory. */
void dump(unsigned int address, unsigned int length)
{
    unsigned int i;          /* Points to the current byte being dumped. */
    char ascii[DUMPHWIDTH+1]; /* Contains the ASCII equivalent of each byte. */

    ascii[DUMPHWIDTH] = NULL; /* Make sure ascii has a null delimiter
                                to look like a C string. */
    /* Convert length to a multiple of DUMPHWIDTH. */
    length = ((length + DUMPHWIDTH-1)/DUMPHWIDTH) * DUMPHWIDTH;
    for (i=0;i<length;i++) {
        if (0==i%DUMPHWIDTH) {          /* Dump the ascii version and start a
                                           new line every DUMPHWIDTH characters. */

```

```

        if (i > 0) {
            printf("%s\n",ascii);
        }
        /* Also, dump the current address. */
        printf("%s: ",uith(address+i));
    }
    /* Put extra spaces in the middle of each line. */
    if (0==i%(DUMPHIDTH/2) && 0 != i%DUMPHIDTH) {
        printf(" ");
    }
    printf("%s ",ctoh(*(char *) (address+i))); /* Dump each byte individually. */
    /* Insert the current character in the string "ascii".*/
    /* If it's not printable, replace it. */
    ascii[i%DUMPHIDTH] = *(char *) (address+i);
    if (ascii[i%DUMPHIDTH] < SPACE || ascii[i%DUMPHIDTH] >= DELETE) {
        ascii[i%DUMPHIDTH] = '.';
    }
}
/* Make sure ascii is printed again at the end of the last line. */
if (i > 0) {
    printf("%s\n",ascii);
}
}

/*****
/* This routine gets a hexadecimal byte from the terminal.*/
*****/
char gethex(void)
{
    int i;
    char string[HSTRLEN + 1];

    string[HSTRLEN] = NULL;
    for (i=0;i < HSTRLEN;++i) {
        string[i] = tolower(termin());
        if (string[i] == BS) {
            i -= 2;
            if (i < -1) {
                i = -1;
            } else {
                printf(" b b");
            }
        }
        continue;
    }
    printf("%c",string[i]);
    if (string[i] >= 'a' && string[i] <= 'f')
        continue;
    if (string[i] >= '0' && string[i] <= '9')
        continue;
    string[i] = NULL;
    break;
}
return(atoi(string));
}

```

```

/*****
/* This routine gets a hexadecimal word (two bytes) from the terminal.*/
*****/
unsigned int gethexint(void)
{
    int i;
    char string[HEXINTSTRLEN+1];

    string[HEXINTSTRLEN] = NULL;
    for (i=0;i < HEXINTSTRLEN;++i) {
        string[i] = tolower(termin());
        if (string[i] == BS) {
            i -= 2;
            if (i < -1) {
                i = -1;
            } else {
                printf("b b");
            }
            continue;
        }
        printf("%c",string[i]);
        if (string[i] >= 'a' && string[i] <= 'f')
            continue;
        if (string[i] >= '0' && string[i] <= '9')
            continue;
        string[i] = NULL;
        break;
    }
    return(atohexint(string));
}

/*****
/* Get an integer from the terminal. */
*****/
int getint(void)
{
    int i;
    char string[STRLEN];

    string[STRLEN] = NULL;
    for (i=0;i < STRLEN;++i) {
        string[i] = termin();
        if (string[i] == BS) {
            i -= 2;
            if (i < -1) {
                i = -1;
            } else {
                printf("b b");
            }
            continue;
        }
        printf("%c",string[i]);
        if (string[i] < '0' || string[i] > '9') {
            string[i] = NULL;
            break;
        }
    }
}

```

```

    return(atoi(string));
}

/*****
int getpageno(void)
{
    int page;
    char s[STRLEN];      /* Storage for itoa(). */
    itoa(MAXPAGE,s);
    printf("Input the bubble memory page number (0-%s decimal): ",s);
    while (TRUE) {
        page = getint();
        printf("n n");
        if (page >= 0 && page <= MAXPAGE)
            break;
        itoa(MAXPAGE,s);
        printf("Page must be in the range (0-%s decimal): ",s);
    }
    return(page);
}

/*****
/* This function checks to see if a character is available through termin().
   It places the character, if any, in the location pointed to by
   'character'. It returns TRUE if there was a character, FALSE otherwise. */
char look_ahead(char *character)
{
    /* If there is no terminal attached, return FALSE. */
    if (!checkprt()) {
        return(FALSE);
    }
    /* If the buffer is already full, return it's contents,
       but don't empty it. */
    if (console_data_available) {
        *character = console_buffer;
        return(TRUE);
    }
    /* Check the RS232C port to see if there is data available.
       Bit 1 will be 1 when data is present. */
    if (input(PRTCTRL) & PRTRDY) {
        /* If there is data, store it in the buffer and let termin() know
           about it by setting the console_data_available flag. */
        *character = console_buffer = input(PRTDATA);
        return(console_data_available = TRUE);
    }
    return(FALSE);      /* No data was present. */
}

/*****
/* This routine checks to see if a character has been entered from the
   keyboard. If so, it discards the character and returns TRUE. If not,
   it returns FALSE. */
char look_ahead_discard(void)
{
    if (look_ahead((char *) NULL)) {

```

```

        termin();
        return(TRUE);
    } else
        return(FALSE);
}

/*****
/* This function obtains a character from the keyboard. */
char termin(void)
{
    static char allow_menu_call = TRUE;

    /* 'allow_menu_call' is true if menu() can be called from termin(),
       FALSE otherwise. It can be called from termin() once; subsequently,
       it must first return control to termin(). Thus, one recursive entry
       into menu() is permitted at a time. The experiment can be monitored,
       but only at one subordinate calling level and no more. */

    char waiting_for_ctls;

    /* This variable is true if an odd number of CTRL-S characters has
       been accepted. No characters can be accepted from the keyboard
       until an even number of them have been received. However, CTRL-Y can
       be accepted, in which case menu() will be called at once. */

    char ctrl_valid_data; /* This is TRUE if look_ahead() already filled
                           the buffer; FALSE otherwise. */

    waiting_for_ctls = FALSE;
    ctrl_valid_data = console_data_available;

    /* Loop continuously until an acceptable character has been received. */
    while (TRUE) {
        /* If the buffer is empty, then wait for a character to be entered.
           It could have been filled by look_ahead(). */
        if (!console_data_available) {
            while (TRUE) {
                /* Check the RS232C port to see if there is data available.
                   Bit 1 will be 1 when data is present. Wait for data. */
                if (input(PRTCTRL) & PRTRDY) {
                    console_buffer = input(PRTDATA);
                    break;
                }
            }
        }

        /* Now that console data has been read, set the availability flag
           FALSE so that if it becomes necessary to read another character,
           you can do so. */
        console_data_available = FALSE;
        switch (console_buffer) {
            case (CTRLS):
                /* Toggle the waiting_for_ctls flag. As long as this flag
                   is true, you can't get out of termin(). */
                waiting_for_ctls = !waiting_for_ctls;
                if (!waiting_for_ctls) && ctrl_valid_data)
                    return(console_buffer);
                break;

```

```

    case (CTRL_Y):
        /* You can execute a CTRL-Y even if a CTRL-S is pending.
           The effect is to cancel the CTRL-S. */
        waiting_for_ctls = FALSE;
        if (allow_menu_call) {
            allow_menu_call = FALSE;
            /* Tell menu() not to start the experiment. This is
               only permissible when main() is the calling function. */
            menu(!EXPERIMENTOK);
            allow_menu_call = TRUE;
        }
        if (ctrl_valid_data)
            return(console_buffer);
        break;
    default:
        /* Ignore this character if you're waiting for a second CTRL-S. */
        if (waiting_for_ctls) {
            break;
        }
        return(console_buffer);
    }
}

/*****
/* This function allows the user to read data from any port. */
void testinput(void)
{
    int port;          /* Port number to be entered from the keyboard.*/
    char data;         /* Data to be read from that port. */

    printf("Specify port address to be read (in hexadecimal): ");
    port = gethex();    /* Get the port address. */
    printf("n r");
    data = input(port); /* Read from the port. */
    printf("Data from port (in hexadecimal): %s n",ctoh(data));
}

/*****
/* This routine outputs a character to a specified port. */
void testoutput(void)
{
    int port;          /* The port address. */
    char data;         /* The data to be sent to the port. */

    printf("Specify port address to be written to (in hexadecimal): ");
    port = gethex();    /* Get the port address. */
    printf("n r");
    printf("Specify the data to be sent to the port (in hexadecimal): ");
    data = gethex();
    output(port,data);
}

```

X. FILENAME MAIN.H

```
/* This file contains external prototyping declarations for all functions
in "main.c". */
```

```
extern void memory_dump(void);
extern char menu(char experiment_flag);
extern void program(void);
extern void testio(void);
extern void main(void);
```

Y. FILENAME MAIN.C

```
/* main.c */
```

```
#include "version.h"
#include "vibro.h"
#include "bubble.h"
#include "inout.h"
#include "power.h"
#include "convert.h"
#include "initial.h"
#include "clock.h"
#include "newio.h"
#include "global.h"
#include "expmnt.h"
```

```
void memory_dump(void);
char menu(char experiment_flag);
void testio(void);
void main(void);
```

```
/*
*****
/* This routine lets the user produce memory dumps for any section of memory.*/
void memory_dump(void)
{
    unsigned int address; /* Will hold the starting address of the dump.*/
    unsigned int length; /* Will hold the number of bytes to dump.*/
    while (TRUE) {
        printf("Please specify address: ");
        address = gethexint();
        printf("\nrPlease specify number of bytes to dump (0 to quit): ");
        length = gethexint();
        printf("\nr");
        if (length == 0)
            break;
        dump(address, length);
    }
}
*****
/* This routine is the highest level of all the diagnostic menus.
It will not permit the experiment to be run unless the experiment_flag
is TRUE. */
char menu(char experiment_flag)
{
```

```

char    data;          /* A character read from the keyboard. */
int     i;             /* A counter. */
char    addata;        /* A value read from the A/D converter. */
int     value;         /* The A/D reading converted to useful units. */

while(TRUE) {
    version();
    printf("\n\n
A Software reset.n r
B Realtime clock functions.n r
C Power relay switching functions.n r
D Bubble memory test functions.n r
E A/D converter functions.n r
F Run experiment.n r
G Perform port I/O.n r
H Display contents of controller memory.n r
I Examine or change the data logged in the bubble memory.n r
Z Exit this menu.n r");

    /* Read in a character from the keyboard, convert it to lower case,
    and display it. */
    data = tolower(termin());
    printf("%c\n",data);
    switch (data) {
        case 'a':          /* To perform a software reset, jump
                             to address 0. */
            asm(" jp    0");
            break;
        case 'b':          /* Call the real time clock functions. */
            rtc();
            break;
        case 'c':          /* Call the power control functions. */
            pwrcnt();
            break;
        case 'd':          /* Call the bubble memory testing
                             functions. */
            bubmenu();
            break;
        case 'e':          /* Display the A/D data. */

            for (i=0;i < ADPOINTS;++i) {
                addata = ad_read(adport[i]);
                printf("%-24s=%3.0d=",adcaption[i],addata);

                /* If i <= 2, then the A/D reading is a voltage. */
                if (i <= 2) {
                    value = adtoint(addata,(i==2)?MULT_10V : MULT_20V);
                    printf("%c%2.0d.%02.0dV  ",(i==1)?'-':'+',
                        value/100,value%100);

                    /* Otherwise, the A/D reading is a temperature. */
                } else {
                    value = adtoint(addata,MULT_TEMP);
                    printf("%6.0dK  ",value);
                }
            }
    }
}

```

```

        /* Print two points per line. */
        if ((i != i % 2) || i == ADPOINTS - 1)
            printf("n");
    }
    break;
case 'f':
    /* Execute the experiment, unless it
       is currently in a suspended state. */
    if (experiment_flag) {
        expmnt();
    } else {
        printf("You cannot run the experiment functions while
execution is suspended with (Y. n rExit this menu and try again. n r");
    }
    break;
case 'g':
    /* Enter the routine which reads from and
       writes to any port. */
    testio();
    break;
case 'h':
    memory_dump();
    /* Enter the routine which displays the
       contents of selected portions of
       memory. */
    break;
case 'i':
    log_menu();
    /* Enter the routine which permits the
       contents of the bubble memory log to
       be modified. */
    break;
case 'z':
    return;
default:
    printf("Use a valid letter please! n r");
}
}

/*****
/* This routine allows you to output data manually to any port, or to read
   data from any port. */
void testio(void)
{
    char    data; /* A character entered from the keyboard. */

    /* Repetitively display the following menu until choice Z is made. */
    while (TRUE) {
        printf(
            "nManual port I/O functions. Pick one! n r,
            A Input. n r,
            B Output. n r,
            Z Return to previous menu. n r");

        /* Read a character from the keyboard, convert it to lower case,
           and display it. */
        data = tolower(termin());
        printf("%c n r", data);
        switch(data) {

```

```

        case 'a':                /* Enter the function which allows the
                                user to read data from a port. */
            testinput();
            break;
        case 'b':                /* Enter the function which allows the
                                user to write data to any port. */
            testoutput();
            break;
        case 'z':                /* Quit. */
            return;
        default:
            printf("Use a valid letter please.\n");
            break;
    }
}

/*****
/* The C program begins here. This routine gets control from the assembly
   language program which resides at address 0. */
void main(void)
{
    /* Make sure that each of these pointers is initialized to point to the
       same memory as the corresponding buffer. Thus the same data can be
       accessed either as a list of characters (a buffer) or as a structure
       (if the contents need to be accessed individually.) */
    pagezero = (struct page0data *) page0_buffer;
    log_page  = (struct full_log_page *) log_buffer;

    /* Initialize the system ports. */
    inithardware();

    /* See if there is a terminal attached. If so, turn off any subsystems
       which are currently on and enter the menu diagnostic system. */
    if (prtconnected == checkprt()) {
        shut_down_no_log();
        while (TRUE) {
            menu(EXPERIMENTOK);
        }
    }
    /* If there is no terminal attached, we must be in space, so run the
       experiment. */
    else
        expmmt(); /* Run the experiment. */
}

```

Z. FILENAME MBRK.S

```

; mbrk.s

; ****
; mbrk() function for use with malloc() and calloc().
; File "spec" declares a single section of RAM, MRAMSZ bytes long,
; to use for memory allocation, and START in file "start.asm"
; initializes MBRKPTR to point to that memory.
; ****

```

```

global mbrk,MBRKPTR,MRAMSZ
option long=4           ; assume long=4 bytes

region code
mbrk: push ix           ; char *mbrk( long size, long *realsize );
ld ix,0
add ix,sp               ; (ix+4,5,6,7):size, (ix+8,9):realsize
ld de,MBRKPTR           ; return value is address of memory section
ld a,d
or e
jr z,out               ; zero means memory section is in use
ld a,(ix+6)
or (ix+7)
jr nz,out              ; nonzero means more than 64K requested
ld c,(ix+4)             ; requested 'size' to bc
ld b,(ix+5)
ld hl,MRAMSZ            ; check if 'size' bytes are available
                        ; assumes MRAMSZ is less than 64K
                        ; clear the carry flag.
or a
sbc hl,bc
jr c,out
ld bc,0                ; mark memory section as used
ld (MBRKPTR),bc        ; de still holds former MBRKPTR
ld l,(ix+8)             ; get the pointer to 'realsize'
ld h,(ix+9)
ld (hl),lo MRAMSZ      ; write back actual size of memory section
inc hl
ld (hl),hi MRAMSZ
inc hl
ld (hl),0
inc hl
ld (hl),0
jr ret                 ; de is the return value
out: ld de,0           ; out of memory, return zero pointer
ret: ld sp,ix
pop ix
ret

```

AA. FILENAME NEWIO.H

```

extern char input(char port);
extern void output(char port,char data);

```

AB. FILENAME NEWIO.S

```

; newio.s

export input, output
region code

; char input(char port);
input:
push ix                ;There are no local variables.
ld ix,0

```

```

add    ix,sp
ld     c,(ix+4)    ;Put port address in register c.
in     a(c)        ;Get the data from the port.
pop    ix          ;Restore ix to the value it had before this
                    ;function was called.

ret

; void output (char port, char data);
output:
push   ix
ld     ix,0        ;There are no local variables.
add    ix,sp
ld     c,(ix+4)    ;Put port address in register c.
ld     a,(ix+6)    ;Put data in register a.
out    (c),a       ;Write the data to the port.
pop    ix          ;Restore ix to the value it had before this
                    ;function was called.

ret

```

AC. FILENAME POWER.H

```

/* This file contains external prototyping declarations for all functions
in "power.c". */

```

```

extern char power_status(void);
extern char power_write(char command);
extern void pwrcnt(void);

```

AD. FILENAME POWER.C

```

/* power.c */

```

```

#include "vibro.h"
#include "bubble.h"
#include "convert.h"
#include "inout.h"
#include "delay.h"
#include "expmnt.h"
#include "newio.h"
#include "global.h"

```

```

char power_status(void);
char power_write(char command);
void pwrcnt(void);

```

```

/*****
/* This routine gets the status from the power board. */

```

```

char power_status(void)
{
    return(input(POWERIN));
}

```

```

/*****
/* This routine sends commands to the power board. */

char power_write(char command)
{
    int i;
    char    status;
    char    oncommand; /* TRUE if command is an ON command,
                        FALSE otherwise. */
    char    relayon; /* TRUE if the indicated relay is on,
                     FALSE otherwise. */
    /* Try to send the command to the power board.
       Return TRUE if successful. Return FALSE after you give up
       in disgust. */
    for(i=0;i<TRIES;i++) {
        output(POWEROUT,command);
        output(BSETC1,PHRSTROBE);
        delay(PWRDELAY);
        output(BCLRCL,PHRSTROBE);
        delay(PWRDELAY); /* Wait PWRDELAY x 10 ms for the
                        relays to respond. */
        /* The command is intended to turn a relay on only if the
           last bit is set (1). */
        oncommand = ONBIT & command;
        status = NOPOWER | power_status();
        /* To see whether the indicated relay is on,
           see whether only the status bit in the
           relay's assigned bit position is a zero. If it is, then
           that relay is on. */
        relayon = status & command;
        /* If the relay's position matches that commanded, then
           log a successful setting of the relay. */
        if ((oncommand && relayon) || (!oncommand && !relayon)) {
            return(TRUE);
        }
        printf("Trying again to switch relays. n r");
    }

    /* If you got this far, then the relay's position did not match
       that commanded, so log a failure. */
    return(FALSE);
}

/*****
/* This routine sends commands to the power board. */
*****/
void pwrent(void)
{
    char    data; /* Data from the keyboard. */

    static int relay[] = {
        SSDRON, SSDROFF, VCOON, VCOOFF, ADON, ADOFF,
        MATFON, MATFOFF, HEATON, HEATOFF
    };

    while(TRUE) {

```

```

        printf("\nPOWER SWITCH CONTROL. n\n\n");
A = SDDR on. n\n
B = SDDR off. n\n
C = VCO on. n\n
D = VCO off. n\n
E = A/D on. n\n
F = A/D off. n\n
G = MATCHED FILTER on. n\n
H = MATCHED FILTER off. n\n
I = HEATER on. n\n
J = HEATER off. n\n
K = READ power status port. n\n
Z = Back to the MAIN MENU. n\n");

    data = tolower(termin());
    printf("%c\n",data);
    if (data >= 'a' && data <= 'j') {
        if(!power_write(relay[data-'a']))
            printf("Power control command failed. n\n");
    } else {
        switch (data) {
            case 'k':
                printf("%s\n",ctoh(power_status()));
                break;
            case 'z':
                return;
            default:
                printf("Use a valid letter, please. n\n");
                break;
        }
    }
}
}
}

```

AE. FILENAME START.S

```

;*****
;
;   February 19, 1988                start.s
;
;   This startup code initializes interrupt vectors and runs START at
;   reset
;   to initialize RAM and call the user function main().
;   The companion link specification file is "spec" which defines
;   many of the imported symbols. Also see file "mbrk.asm" for the
;   mbrk() function if you want to use malloc() or calloc().
;   The program is adapted from an example given in the UNIWARE
;   manual, Compiler section, pp. 13-15.
;*****
export  START,MBRKPTR
import  main,STACKTOP,RAMDATA,ZRAM,ZRAMSZ,IRAM,IRAMSZ,MRAM

```

```

;*****
;   Define a variable to track memory allocations in mbrk().
;*****
        region ram
MBRKPTR ds 2                ; (char *) to available memory

;*****
;   Reset code must be linked to address 0.
;*****
        region reset
        ld sp, 10 STACKTOP ; initial stack pointer (0x10000 as 0)
        jp START           ; initial execution address

ARESTART:                ;RESTART LOCATION 1
        jp START
        org 0x10
BRESTART:                ;RESTART LOCATION 2
        jp START
        org 0x18
CRESTART:                ;RESTART LOCATION 3
        jp START
        org 0x20
DRESTART:                ;RESTART LOCATION 4
        jp START
        org 0x28
ERESTART:                ;RESTART LOCATION 5
        jp START
        org 0x2C
FRESTART:                ;RESTART LOCATION C
        jp START
        org 0x30
GRESTART:                ;RESTART LOCATION 6
        jp START
        org 0x34
HRESTART:                ;RESTART LOCATION B
        jp START
        org 0x38
IRESTART:                ;RESTART LOCATION 7
        jp START
        org 0x3C
JRESTART:                ;RESTART LOCATION A
        jp START
        org 0x66
NONMASKI:                ;NON-MASKABLE INTERRUPT
        jp START

;*****
;   This code can be anywhere; the reset code jumps to it.
;*****
        region code
START: ld ix,0            ; end of stack frame chain
        ld hl,MRAM        ; initialize memory allocator
        ld (MBRKPTR),hl

;*****
;   Zero out uninitialized RAM.

```

```

; It is assumed here that ZRAMSZ > 1 but this is guaranteed
; as long as MBRKPTR (above) is defined in region ram.
;*****
ld    hl,ZRAM          ; zero ZRAMSZ bytes here
ld    (hl),0           ; zero first byte
ld    de,ZRAM+1        ; repeatedly zero other bytes
ld    bc,ZRAMSZ-1
ldir

;*****
;      Initialize other RAM from ROM.
;*****
ld    hl,RAMDATA
ld    de,IRAM
ld    bc,IRAMSZ
ld    a,b
or    c
jr    z,none
ldir
none:

;*****
;      Invoke main() with no arguments.
;*****
call   main            ; any return value is "int" in de
done:  halt            ; halt if main returns

;*****
; To vector an interrupt to a C function, you must go though
; a register save routine like the one shown here.
; If the "-r exx" option is being given to the command line,
; then registers bc' de' and hl' need not be saved and restored
; since the compiler will make no use of them. The compiler
; does not use af' in any case.
;*****
      region    code
;INTERRUPT
;  push    af          ; save registers
;  push    bc
;  push    de
;  push    hl
;  push    ix
;  push    iy
;  exx
;  push    bc
;  push    de
;  push    hl
;  exx
;  call    cfcn        ; call some C function
;  exx
;  pop     hl          ; restore registers
;  pop     de
;  pop     bc
;  exx
;  pop     iy
;  pop     ix
;  pop     hl

```

```

} pop    de
} pop    bc
} pop    af
} ei
} ret    ; return from interrupt

```

AF. FILENAME ASM.BAT

```

@rem    Make asmsource the current subdirectory
cd vibro contrlr asmsource
@rem    Assemble the specified source file
uas280 -c 80 -n -t 4 -L %1
@rem    Place the object module in the object subdirectory
copy *.o vibro contrlr object
erase *.o
@rem    Place the assembly listing in the list subdirectory.
copy *.lst vibro contrlr list
erase *.lst

```

AG. FILENAME ASMLIST.BAT

```

@rem    Fill in the symbols of the specified assembly listing file
@rem    with the values given in the executable module u.out.
@rem    Pipe the completed listing to the ulist program to give a
@rem    decent looking print-out.
uabs vibro contrlr u.out < vibro contrlr list %1 | ulist >> temp print

```

AH. FILENAME C.BAT

```

@rem    Make csource the current subdirectory.
cd vibro contrlr csource
@rem    Compile the source file.
ucc280 -e -l -A -L -- %1
@rem    Place the resultant object module in the object subdirectory.
copy *.o vibro contrlr object
erase *.o
@rem    Place the resultant assembly listing in the list subdirectory.
copy *.lst vibro contrlr list
erase *.lst

```

AI. FILENAME LINK.BAT

```
@rem Make object the current subdirectory.
cd vibro contrlr object
@rem Link the specified object modules together.
ule -f spec -t -v %1 %2 %3 %4 %5 %6 %7 %8 %9
@rem Place the linked module in the contrlr subdirectory as u.out.
copy *.out vibro contrlr
erase *.out
@rem Create an executable module in the contrlr subdirectory as u.bin.
cd vibro contrlr
ufihex u.out > vibro.hex
```

AJ. FILENAME LIST.BAT

```
@rem Produce a paginated listing of the specified file, and
@rem put it in a temporary, scratch file called temp print.
ulist -d -t 4 -x -O hdr=%1 %1 >> temp print
```

AK. FILENAME LOADMAP.BAT

```
@rem Create a load map of all the regions in u.out.
unm -m vibro contrlr u.out > temp temp
@rem Produce a paginated print-out of it.
ulist -d -O hdr=loadmap temp temp >> temp print
```

AL. FILENAME PRINTALL.BAT

```
@rem Produce a complete listing of the load map, symbol table and
@rem all source files, and header files.
cd vibro contrlr
call readyout
call loadmap
call promsym
call o
call list spec
call h
call list version.h
call cs
call list version.c
call h
call list vibro.h
call list bubble.h
call cs
call list bubble.c
call h
call list bubrw.h
call s
call list bubrw.s
call h
call list clock.h
```

```

call cs
call list clock.c
call h
call list convert.h
call cs
call list convert.c
call h
call list delay.h
call s
call list delay.s
call h
call list expmnt.h
call cs
call list expmnt.c
call h
call list global.h
call cs
call list global.c
call h
call list initial.h
call cs
call list initial.c
call h
call list inout.h
call cs
call list inout.c
call h
call list main.h
call cs
call list main.c
call s
call list mbrk.s
call h
call list newio.h
call s
call list newio.s
call h
call list power.h
call cs
call list power.c
call list s.bat
call list start.s
call b
call list asm.bat
call list asmlist.bat
call list b.bat
call list backup1.bat
call list backup2.bat
call list c.bat
call list cs.bat
call list link.bat
call list list.bat
call list loadmap.bat
call list o.bat
call list printall.bat
call list promlib.bat
call list promlink.bat

```

```
call list promsym.bat
call list readyout.bat
```

AM. FILENAME PROMLINK.BAT

```
cd vibro contrlr object
link -F linkfile libc.a
```

AN. FILENAME PROMOUT.BAT

```
@rem Put the print scratch file into the printer queue.
copy vibro contrlr batch lpfont+ temp print+ vibro contrlr batch normfont temp print2
print temp print2
```

AO. FILENAME PROMSYM.BAT

```
@rem Put the symbols from u.out into a scratch file.
unm -fmrstv#gx vibro contrlr u.out > temp temp
@rem Produce a paginated version of the symbol table listing.
ulist -d -O hdr=symbols temp temp >> temp print
```

AP. FILENAME READYOUT.BAT

```
@rem Get rid of the two scratch files used in producing listings.
erase temp temp
erase temp print
erase temp print2
```

APPENDIX I. RS-232C INTERFACE PIN CONNECTIONS

This appendix contains the complete electrical specification for the RS-232C Interface. It is provided here for convenience. Only a subset of this specification has been implemented in the Vibro-acoustic Experiment for the purpose of providing communications between the controller and the terminal, which is useful during ground testing.

Table 16. RS-232C INTERFACE PIN CONNECTIONS

Pin Number	Circuit Designation	Mnemonic Designation	Direction	Description
1	AA	FG		FRAME GROUND. This lead is an electrical equipment frame and power ground.
2	BA	TD	To DCE	TRANSMITTED DATA. This lead carries the serial digital data transmitted from the DTE to the DCE.
3	BB	RD	To DTE	RECEIVED DATA. This lead carries the serial digital data received at the DTE.
4	CA	RTS	To DCE	REQUEST TO SEND. An "ON" condition on this lead is used to enable the local DCE for data transmission.
5	CB	CTS	To DTE	CLEAR TO SEND. An "ON" condition on this lead indicates whether or not the DCE is ready to transmit data.
6	CC	DSR	To DTE	DATA SET READY. An "ON" condition on this lead indicates that the local DCE is ready to process data and is not in a test, talk, or dial mode.

Source: Couch, L. W., *Digital and Analog Communication Systems*, Macmillan Publishing Company, 1987, pp. 684-686

Table 17. RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)

Pin Number	Circuit Designation	Mnemonic Designation	Direction	Description
7	AB	SG		SIGNAL GROUND. This lead establishes the common ground reference potential for all circuits except frame ground on pin 1.
8	CF	DCD	To DTE	DATA CARRIER DETECT (Received Line Signal Detector). This lead indicates that data from the remote location is being received and meets a suitable criterion established by the DCE manufacturer.
9			To DTE	Positive DC Test Voltage
10			To DTE	Negative DC Test Voltage
11	Bell 208A type circuit	QM	To DTE	EQUALIZER MODE. This lead is used to indicate to the DTE that the adaptive equalizer in the receiver is reset automatically when error performance is poor. (non-EIA designated).
12	SCF	(S)DCE	To DTE	SECONDARY DATA CARRIER DETECT. This lead is equivalent to DCD on pin 8 except that it indicates the proper reception of the secondary channel line signal instead of the primary channel received line signal.
13	SCB	(S)CTS	To DTE	SECONDARY CLEAR TO SEND. This lead is equivalent to CTS on pin 5 except that it indicates the availability of the secondary channel instead of indicating the availability of the primary channel to transmit data.

Source: Couch, L. W., *Digital and Analog Communication Systems*, Macmillan Publishing Company, 1987, pp. 684-686

Table 18. RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)

Pin Number	Circuit Designation	Mnemonic Designation	Direction	Description
14	SBA	(S)TD	To DCE	SECONDARY TRANSMITTED DATA. This lead is equivalent to TD on pin 2 except that it is used to transmit data via the secondary channel.
	Bell 208A type circuit	NS	To DCE	NEW SYNC. This lead may be used on an optional basis with the DCE at a master station of a multistation private line network, such as in a polling operation, to ensure rapid resynchronization of the receiver on data from many different remote transmitters (non-EIA designated).
15	DB	TC	To DTE	TRANSMITTER CLOCK. This lead is used to provide the DTE with signal element timing information.
16	SBB	(S)RD	To DTE	SECONDARY RECEIVED DATA. This lead is equivalent to RD on pin 3 except that it is used to receive data on the secondary channel.
	Bell 208A type circuit	DCT	To DTE	DIVIDED CLOCK, TRANSMITTER. A square-wave signal at one-third the nominal bit rate appears on this lead whenever power is supplied to the DCE (non-EIA designated.).

Source: Couch, L. W., *Digital and Analog Communication Systems*, Macmillan Publishing Company, 1987, pp. 684-686

Table 19. RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)

Pin Number	Circuit Designation	Mnemonic Designation	Direction	Description
17	DD	RC	To DTE	RECEIVER CLOCK. This lead is used to provide the DTE with received signal element timing information.
18	Bell 208A type circuit	DCR	To DTE	DIVIDED CLOCK, RECEIVER. A square-wave signal on this lead provides the receiver timing information at one-third the nominal bit rate (non-EIA designated).
19	SCA	(S)RTS	To DCE	SECONDARY REQUEST TO SEND. This lead is equivalent to RTS on pin 4 except that it requests to use the secondary channel instead of the primary data channel.
20	CD	DTR	To DCE	DATA TERMINAL READY. An "ON" condition on this lead indicates that the DTE is ready to be connected to the communication channel.
21	CG	SQ	To DTE	SIGNAL QUALITY DETECT. This lead is used to indicate whether or not there is a high probability of an error in the received data.

Source: Couch, L. W., *Digital and Analog Communication Systems*, Macmillan Publishing Company, 1987, pp. 684-686

Table 20. RS-232C INTERFACE PIN CONNECTIONS (CONTINUED)

Pin Number	Circuit Designation	Mnemonic Designation	Direction	Description
22	CE	RI	To DTE	RING INDICATOR. An "ON" condition on this lead indicates that a ringing signal is being received on the communication channel.
23	CH		To DCE	DATA RATE SELECTOR. This lead is used to select between the two data signalling rates in the case of dual rate DCE.
	CI		To DTE	DATA RATE SELECTOR. This lead is used to select between the two data signalling rates in the case of dual rate DCE.
24	DA	TC	To DCE	EXTERNAL TRANSMITTER CLOCK. This lead is used to provide the transmitting signal converter with signal element timing information.
25	Bell 208A type circuit		To DCE	BUSY. This lead is used for testing purposes by Telephone Company personnel (non-EIA designated).

Source: Couch, L. W., *Digital and Analog Communication Systems*, Macmillan Publishing Company, 1987, pp. 684-686

LIST OF REFERENCES

1. INTEL Corporation, *BPK 5V75A Four-Megabit Bubble Memory Prototyping Kit User's Manual*, No. 2444-001, (undated).
2. Couch, Leon W., *Digital and Analog Communication Systems*, Macmillan Publishing Company, 1987.
3. Wallin, J. W., *Microprocessor Controller with Nonvolatile Memory Implementation*, MSEE Thesis, Naval Postgraduate School, Monterey, CA, December 1985.
4. G.A.S., *Small Self-contained Payloads, Experimenter Handbook*, National Aeronautics and Space Administration, Goddard Space Flight Center, 1987.
5. Stehle, C. D., *Vibration Isolation of a Microphone*, MS in Engineering Acoustics Thesis, Naval Postgraduate School, Monterey, CA, September 1985.
6. Jordan, D. W., *A Matched Filter Algorithm for Acoustic Signal Detection*, MSEE Thesis, Naval Postgraduate School, Monterey, CA, June 1985.
7. Boyd, A. W., Kosinski, B. P., and Weston, R. L., "Autonomous Measurement of Space Shuttle Payload Bay Acoustics During Launch," *Naval Research Reviews*, Vol. 39, No. 1, pp. 9-17, 1987.
8. Frey, T. J., Jr., *A 32-Bit Microprocessor Based Solid State Data Recorder for Space Based Applications*, MSEE Thesis, Naval Postgraduate School, Monterey, CA, March 1986.
9. Kuebler, D. P., *Signal Acquisition and Processing for Autonomous Space Shuttle Cargo Bay Acoustic Measurements*, Defense Technical Information Center (DTIC) Report No. ADA200426, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1988.

10. National Semiconductor Corp., *NSC800 High-Performance Low-Power Micro-processor*, July 1983.
11. National Semiconductor Corp., *NSC810A RAM-I/O-Timer*, February 1984.
12. Ghausi, M. S., and Laker, K. R., *Modern Filter Design*, Prentice-Hall, Inc., 1981.
13. *Micro-Cap III Electronic Circuit Analysis Program Instruction Manual*, First Edition, Spectrum Software, 1988.
14. Jung, W. G., *IC Op-Amp Cookbook*, Third Edition, pp. 236-237, Howard W. Sams & Company, 1986.
15. S. Michael, *Notes for EC4100 (Advanced Network Theory)*, Naval Postgraduate School, Monterey, CA, 1988 (unpublished).
16. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., 1978.
17. Software Development Systems, Inc., *UNIWARE Software Development System, Release 3.2*, 1986.
18. Bilofsky, W., *TOOLWORKS C.80*, Version 3.1, The Software Toolworks, 1984.
19. National Semiconductor Corp., *Linear Databook*, 1982.
20. *PCPP PC Personal Programmer User's Guide*, Revision-002, Change 1, Intel Corporation, 1987.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Commander Naval Space Command Attn: Code N3 Dahlgren, VA 22448	1
4. Commander United States Space Command Attn: Technical Library Peterson AFB, CO 80914	1
5. Navy Space System Division Chief of Naval Operations (OP-943) Washington, DC 20305-2000	1
6. Department Chairman, Code 62 Dept. of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
7. Dr. Rudolf Panholzer Chairman, Space Systems Academic Group Code 72 Naval Postgraduate School Monterey, CA 93943	2
8. Mr. Larry Frazier Naval Postgraduate School Monterey CA 93943-5000	1
9. Dr. Sherif Michael Dept. of Electrical and Computer Engineering Code 62Mi Naval Postgraduate School, Monterey, CA 93943	1

10. National Aeronautics and Space Administration 2
Technical Library
NASA Headquarters
600 Independence Ave.
Washington, DC 20546
11. Mr. David Rigmaiden, Code 72 1
Space Systems Academic Group
Naval Postgraduate School
Monterey, CA 93943-5000
12. LT Charles B. Cameron, USN 2
1139 Leahy Rd.
Monterey, CA 93940-5318
13. Prof. Steven Garrett, Code 61Gx 2
Dept. of Physics
Naval Postgraduate School
Monterey, CA 93943-5000
14. Prof. Tom Hofler, Code 61Hf 1
Dept. of Physcis
Naval Postgraduate School
Monterey, CA 93943-5000
15. Space Projects Group, Code 72 3
Naval Postgraduate School
Monterey, CA 93943-5000
16. CDR Steven P. Hannifin USN 1
c o Carrier Airborne Early Warning Squadron 110
NAS Miramar, CA 92145-5000
17. CDR R. Braden, USN 1
c o Carrier Airborne Early Warning Squadron 110
NAS Miramar, CA 92145-5000
18. CPT R. Byrnes, USA 1
c/o Code 39
Naval Postgraduate School
Monterey, CA 93943-5000
19. Research Administration (Code 012) 1
Naval Postgraduate School
Monterey, CA 93943
20. LT Stewart Cobb 1
SSD CLFPD
P. O. Box 92960
LLAFB
Los Angeles, CA 90009-2960

21. Office of Naval Research 1
Physics Division - Code 1112
800 N. Quincy St.
Arlington, VA 22217
22. Commanding Officer 1
Naval Research Laboratory
Attn: E. Senasack (Code 8220)
4555 Overbrook Ave.,
Washington, DC 20375-5000

Thesis

C1925 Cameron

c.1 Control of an experiment to measure acoustic noise in the space shuttle.

24 AUG 90

JUN 16 92

3 2 1 8 4

Thesis

C1925 Cameron

c.1 Control of an experiment to measure acoustic noise in the space shuttle.



thesC1925

Control of an experiment to measure acou



3 2768 000 90767 9

DUDLEY KNOX LIBRARY